THE MICRO WORKS

MACRO-80C

ASSEMBLER REFERENCE MANUAL

P.O. BOX 1110 DEL MAR, CA 92014 714-942-2400

M A C R O - 8 0 c

Assembler Reference Manual

Manual and Software
Written by
Andrew Phelps
The Micro Works

MACRO 80c

Assembler Reference Manual

Disk Based Macro Assembler for
Radio Shack's Color Computer

from
The Micro Works, Inc.

Written by
Andrew E. Phelps

THE MICRO WORKS, INC.

Mailing Address:
P O Box 1110
Del Mar, CA 92014

UPS shipping address:
1942 S. El Camino Real
Encinitas, CA 92024

MACRO-80c

The Micro Works Disk Based Macro Assembler for the Color Computer

TABLE OF CONTENTS

INTRODUCTION

This manual describes MACRO-80c, a macro assembler for the Color Computer. This assembler is an easy-to-use, yet powerful tool for the assembly language programmer. Its purpose is to read and assemble a file or files which have been written using a text editor. This is the "source program" in 6809 assembly language. It produces a file which is the "object program" in machine language. It also produces a "source listing" to a printer or to the screen which shows both the assembly and machine language and is used in debugging your program.

Assembly language is the language in which you can directly control the operation of your Color Computer. You can write programs that run thousands of times faster, take up less memory, and still do more than the same program written in Basic. The only catch is that it is harder to learn, but with a few good books and a lot of experimentation any programmer can discover this wonderful world in which almost anything can be done.

For those already experienced in using assemblers (if you're still reading this far) KEEP READING! You'll be surprised. This assembler has a lot of interesting features which you will eventually want to take advantage of. Some may sound far-fetched at first, but you will find that they are there because they are useful.

LIMITED WARRANTY

Do not write on the disk! If any files are added to the disk, there is a possibility that the directory will be miswritten and that the disk will no longer be useable. Therefore, writing to the disk will void your warranty!

If the disk is folded, spindled, or mutilated when it arrives, save all packing material and contact the carrier immediately. If the disk simply cannot be read, return it to us within ninety (90) days for a working copy. See the title page for our addresses.

The programs on the disk are sold as is without any warranty. We believe the software to operate as described, but can make no representation that it will. We appreciate hearing about bugs, and will make every effort to correct them in future versions. See the section below about the update service.

SOFTWARE LICENSE AGREEMENT

Guess what - you didn't buy the assembler! You bought a disk and a manual, but the assembler program you just bought a license to use. This means that you CAN'T GIVE IT AWAY to your friends, or sell it, or lend it, or transmit it to your favorite bulletin board, or ANYTHING except just use it. It is copyrighted, you know. This is the deal: We're not copy-protecting the disk, so your life is easier because you can back it up to as many of your own disks as you like, and if people don't rob us blind, we'll continue to sell other useful programs on copyable disks.

REGISTRATION

We have included an Owner's Registration form, which you will want to fill out and send in. This does three things for you. (1) You will be informed of new products from The Micro Works, and any information about this assembler which may be sent out. (2) You will be eligible for the update service described below. (3) You will be identified as a bona fide assembler owner should you wish to contact The Micro Works for any reason.

Do not send in a photocopy of the registration form. You must send in the original form.

UPDATE SERVICE

There will probably be revisions to this assembler from time to time. If you send in your Owner's Registration form, you will be informed of these. Then, for a small update fee, you may send in your old disk for a copy of the new version of the program.

SECTION 1 - THE BASICS

RUNNING THE ASSEMBLER

The first thing to do is to write-protect the disk! Do not add any files to the assembler disk, as any updates to the directory could be miswritten and render the disk useless. Now do a BACKUP to another disk, and store away the original in a safe place.

To run the assembler, you must first have something to assemble. This is called the "source program". It is a disk file which has been written using a text editor; refer to your text editor manual for instruction on doing this. For trying out the assembler, you may want to use one of the several example programs included on your disk; these are the ones with names ending in "/TXT".

To run the assembler, type RUN"AS (note that the ending quote is not needed). This is a short Basic program which will automatically load file AS%/BIN from drive 0 into the graphics pages. AS will ask for an input file name. If the file name ends in "/TXT", you do not need to type in the "/TXT" as that will be assumed.

You will next be asked for a binary file name. This is what the assembler should name the output file or "object file". It is often the same name as the input file, only with "/BIN" instead of "/TXT". If you leave off the "/BIN", it will be assumed. If you do not wish to create a binary file and are only assembling for the listing or to check for errors, then just hit ENTER.

The assembler will next ask for options. If all you want to do is to produce a binary file, simply hit ENTER. The options LDS will get you a listing and symbol table; the "D" stands for "delay" and will slow down the screen output. To get a listing to the printer, see the section on "LISTING TO A PRINTER". For other options, see the section on "ASSEMBLER OPTIONS".

After entering the options, the assembler will run. Pressing BREAK will abort the assembly; the spacebar will stop the listing and allow you to single step. Remember that the keyboard is not being scanned during disk I/O so you may have to hold down a key for a second or so. "S" and "F" will slow down or speed up a listing, and pressing them repeatedly will slow it down or speed it up even more.

If a file already exists by the same name as the binary file which is to be created, the message appears, "OLD BINARY WILL BE DELETED". If you then realize that you do not wish the old file deleted, press BREAK and hold it until the assembler stops, since the old file will not be removed before the assembler starts pass 2.

The interaction with AS can easily be modified, as this portion of the assembler is in Basic. For example, you could make a version of AS which always gives the binary file the same name as the input file, or a version which always selects certain options.

The example below shows what will appear on the screen when you assemble the sample program DISPLAY.

```
RUN"AS
WORLD'S BEST ASSEMBLER
(C) 1982 THE MICRO WORKS
INPUT FILE?  DISPLAY
BINARY FILE?  DISPLAY
OPTIONS?
  ****************************
  **  6809 MACRO ASSEMBLER  **
  **    BY ANDREW PHELPS    **
  **(C) 1982 THE MICRO WORKS**
  ****************************
PAGE 0001    DISPLAY
NO ERRORS FOUND
ASSEMBLY DONE
OK
```

ASSEMBLER OPTIONS

When the assembler asks for options, any of the following may be entered (in any order). Any other character in the option string will generate a message and be ignored by the assembler.

- L Listing. If this is not specified, only error lines are listed.
- S Symbol table. This gives a sorted table of symbols and their values. Local labels are not included in the symbol listing.
- X Cross reference. This overrides option S. Each symbol is listed with all line numbers where it is referenced.
- M List macro expansions. This is useful for debugging macros; normally only calls to macros are listed.
- W Long branch warning. After debugging a program, use this option to generate an error wherever a short branch could be used in place of a long one.
- Z Zero byte error. See the section on embedding machine language in Basic programs.
- E Error halt. This puts the assembler into single step mode whenever an error is encountered.
- D Delay output. This slows down the listing to the screen. It may be sped up again by pressing "F" several times once the listing starts.

EXECUTING MACHINE LANGUAGE PROGRAMS

After you have run the assembler, you now have a machine language file on disk. There are several ways of running it.

The usual way of running a machine language file is with LOADM and EXEC. The statement LOADM"filename" will load the file into memory, and EXEC will run it. If you put a parameter after the word EXEC, you can start execution at any memory location; otherwise the "transfer address" of the file (usually the beginning of the file) is where execution starts.

While debugging a program, you may wish to load both it and a machine language monitor (such as DCBUG from The Micro Works). For example, if you are loading a program at $0E00 such as DISPLAY, you would use an offset value in loading DCBUG which was at least the length of DISPLAY. Since DISPLAY has a length of $009E, you could type:

```
LOADM"DISPLAY
LOADM"DCBUG",&H0100
EXEC
```

You would now find yourself in DCBUG, where you could examine and change memory before jumping to DISPLAY with the J command.

The assembler will start the machine language file at location $0E00 unless told otherwise with an ORG statement. This will put the file at the start of the graphics pages, which is a good place for them. If the usual four graphics pages are allocated, this gives machine language 6K of space (up to $2600) before it runs into a Basic program. Even if it does run into Basic, however, this is no problem unless you are loading and calling the program from within a Basic program. If you type NEW after running the machine language, this will reset Basic's pointers.

Warning: If you allocate more space for disk buffers (eg, FILES 3) then the graphics pages start at $0F00 or beyond, and a program loading at $0E00 could cause problems.

Another note: If a program uses the keyboard input buffer (the area from $02DC through $03D5) it should clear it out again to avoid getting a syntax error when returning to Basic. This is why: If you type EXEC in immediate mode, the EXEC token is run before returning to check if it is the end of line. After all, the statement EXEC:PRINT A is perfectly valid. A good example of clearing the buffer (actually only a few bytes need to be cleared) is given in the example program DISPLAY. Of course, if you never intend to return to BASIC, then this is not a problem.

Another way of executing a machine language file is to embed it in a Basic program. See the section EMBEDDING MACHINE LANGUAGE IN BASIC.

Machine language may be placed above the stack. This is where Radio Shack thinks it ought to be. To do this, use the CLEAR statement to reserve some space, and then load it there and execute it with EXEC

or USRn. The problem is that this is in a different location depending on how much memory is in your computer, so a program which loads at the top of memory in a 16K machine may restrict a 32K machine to only have 16K available.

Whatever method of loading and running your machine language program is used, it is a good idea to write in Position Independent Code (PIC) so that the file may be "offset loaded" and run somewhere else. PIC is code which uses no absolute references into itself so that it does not matter where it is in memory as it runs. See the section in this manual on PIC for tips on how it is written.

LISTING TO A PRINTER

If you wish to have the output of the assembler printed, type the command RUN"ASP instead of RUN"AS. This version of the Basic driver program sends the output to the printer; use options (such as L and S) to tell the assembler what sort of output to generate. If no options are given, only the error messages will be printed.

The program ASP may be customized for your printer. There is a line in the program which looks something like this:

```
R$ = CHR$(46)+CHR$(66)+CHR$(8)+CHR$(80)+"8C"+CHR$(0)
```

The first number is the unit number plus 48. (The 48 is the ASCII code for zero, so CHR$(48) can be written "0"). The 46 therefore is unit -2, which is the printer. The second number is the total number of lines per page. The 8 is the number of blank lines to leave at the bottom of the page, for skipping bottom and top margins. The 80 means an 80-column printer; change this for printers of different widths. The "8C" is the option string; the following options are available:

- 8 Eighty-column format (ie, data and source on same line). Use this for any printer at least 64 columns wide.
- R Suppress carriage return on last column (such as for a 40 column printer which automatically does a carriage return after the 40th column).
- T Truncate listing after last column; otherwise the line will continue on the next line.
- N No paging. The "Number of lines per page" has no effect, and there are no page headers.
- F Use form feed characters ($0C) to get to the top of form. Use this if it works on your printer.
- L Send line feeds. Use this if your printer does not do automatic line feeds.
- C Tab to the comment field. This is normally done with 80 column printers.

LISTING TO A FILE

The listing produced by the assembler may be sent to a file as well as to the screen or to a printer. As indicated in the section LISTING TO A PRINTER, the listing unit as well as other characteristics of the listing are specified by string R$ in the programs AS and ASP. If the first byte of R$ (which is "0" for the screen and CHR$(46) for printer) is set to "1", the listing will be sent to unit 1. Add the statement OPEN"O",#1,"LISTING/TXT" to ASP so that unit 1 will be open when the assembler is called.

As with Basic programs, the number of files that may be open simultaneously is set by the **FILES** statement. You will get an error message if too many files are opened. If you are listing to a file and writing a binary file while the input file is open, you may need to type FILES 3 before running the assembler.

PAGE HEADERS

Unless option "N" is given in the printer setup string (meaning "no pages"), the listing will be divided into pages of the length and width given by the printer setup string. Each page starts with a page number, followed by a page header string. This string is given by the Basic driver program in the line which says A$ = ... and may be modified for your own page headers.

An example of a more elaborate version of this statement might be:

```
A$ = CHR$(14)+"  YOUR NAME HERE  "+CHR$(15)+"^"+CHR$(13)
```

The 14 and 15 are codes for expanded print and normal print on some printers; refer to your printer manual. The 13 at the end is a carriage return; this leaves a blank line after the header line. The up arrow is the signal to print the program name at that position. The program name is specified by the NAM pseudo-op, and inserted in place of the up arrow at the top of each page.

The number of lines of source code printed on each page is determined by the total number of lines per page given, minus the number of blank lines at the bottom, minus the number of lines taken up by the header. Note that if the printer setup string tells the assembler to use a form feed to go to the top of the next page, a smaller number should be given for "total number of lines per page" since lines skipped by form feeds can't be counted.

To print fewer lines on a certain page, use the PAGE pseudo-op. This will cause the rest of the page to be skipped and a new header printed, unless the listing are already just at the top of a new page.

If there is any output during Pass 1 (such as from a MSG pseudo-op), the listing will go to the top of the next page when starting Pass 2. That is why the listing of sample program EX-SQR given later in this manual has a page number 2.

SUMMARY OF PSEUDO-OPS

Your main method of communication with the assembler is by pseudo-ops. These are statements which are placed in the source program along with the statements to be converted to machine language, and they direct the assembler to do such things as generate data, define symbols, set options, or control the listing.

Many of these pseudo-ops are standard ones which are familiar to any 6809 programmer, while others are unique to this assembler. The following is a list of all pseudo-ops recognized by this assembler.

NAM - Stores the name of the program, so that it may be printed in the page headers of the listing. The name may be 0 to 8 characters long and is terminated by a space or end of line. This statement usually appears at the start of every program but it is not required.

END - Signals the end of a source file. If multiple source files are being assembled, there may be multiple END statements. If there is an operand, the value of that expression is saved as the "transfer address" where execution will begin if the object file is run.

INCL - Include source file. The operand field contains the name of a source file which is to be included at this point. Assembly will continue with the next instruction only after the end of the included file is reached.

SETDP - This instruction tells the assembler what value is in the DP register, so that direct addressing may be used. It usually follows an instruction such as TFR A,DP. If no SETDP instruction has been encountered, the value zero is assumed.

ORG - This instruction tells the assembler where to generate code. It may appear any number of times, and there are no restrictions about where and in what order code may be generated. If code is generated without an ORG statement, it starts at location $0E00.

REORG - This instruction resets the assembler's code generation pointer to the value it had just before the last ORG statement. It is used to continue assembly after some specification which required an ORG.

RMB - reserve memory bytes. The operand of this instruction is a number of bytes which are skipped over by the assembler. No code is generated, and the area is not altered when the resulting object file is loaded.

EQU - The operand of this statement is assigned as the value of the symbol name in the label field. In EQU instructions (as well as in ORG, RMB, or SET) no forward reference is allowed in the operand field, as it is necessary to create a correct symbol table during pass 1.

SET - This is syntactically the same as EQU, but allows the symbol value to be redefined. SETs and EQUs may not be mixed with the same symbol.

ASK - This statement allows a label to be set with a value which is typed at the keyboard during pass 1. The operand field contains a string enclosed in quotes; this string is typed to the screen and an expression is typed in by the operator. Example statement:

```
RAMLOC ASK "WHERE IS THE RAM?"
```

MSG - This statement types the values of symbols to the screen during pass 1. The operand field contains a mix of expressions and quoted strings, separated by commas. Examples:

```
MSG "THIS PROGRAM IS ",*-START," BYTES LONG"
MSG "NOW ASSEMBLING SECTION 3"
```

Numeric output is in hexadecimal.

FAIL - This generates an error message. It is generally used within conditional assembly in a macro skeleton, to indicate incorrect macro parameters. The operand field is listed and is used to specify the nature of the error; for example: FAIL "OPERAND TOO BIG"

NOZER - This statement has no operand. Its purpose is to set an option which causes an error to be generated whenever a zero byte is generated. It is used in programs which are to be embedded in lines of BASIC programs, where null bytes are not allowed.

FCC - Form constant character. This generates ASCII data. There are two forms of this statement: a delimited string, and a numbered string. In the numbered string form, there is an expression, a comma, and text. Example: FCC 13,TESTING 1 2 3
In the delimited string form, any character (other than 1-9) is used to start and end the string. Example: FCC "TESTING 1 2 3"

The delimited string may be followed by expressions, in which case the statement becomes an FCB statement. No further strings are allowed in the statement. Example: FCC "TESTING 1 2 3",$0D,0

FCCS - Form constant character in screen code. Like FCC only uses the Color Computer's screen code as shown in the table in this manual.

FCB - Form constant byte. Expressions are separated by commas, and each expression given generates a byte of object code. A byte or series of bytes may be repeated, as in FCB 100[0,1,2] .

FDB - Form double byte. Expressions are separated by commas, and each expression given generates two bytes of object code.

BSZ - Block store zeroes. The operand is a single expression which specifies how many zero bytes to generate. This statement is syntactically identical to RMB, but presets the reserved area to zeroes.

CWORD - Generate checkword. A running 16-bit total of the bytes generated is kept by the assembler, and this total can be inserted into the object code with the CWORD instruction. See the section "USING CHECKSUMS" for a description of how it is used.

CLRC - Clear checkword. If the checkword area (see CWORD above) is not to start with the beginning of the program, the checkword total

may be cleared with CLRC.

LONGVR - Allow use of symbols longer than six characters. (Only the first six letters are used by the assembler, however.)

MACR, ENDM - See section on macros.

IFEQ, IFNE, IFGT, IFGE, IFLT, IFLE,
IFC, IFNC, ELSE, ENDC - See section on conditional assembly.

SPC - The operand of this statement is an expression which gives the number of blank lines to leave in the listing. This is used to make a source listing more readable.

PAGE - This causes the listing to be continued at the top of the next page. There is no effect if the listing already is at the top of a page.

NLST - This suppresses the listing of a section of the program.

LIST - This resumes listing after a NLST. When the number of NLST's encountered exceeds the number of LIST's encountered, listing is suspended; therefore a LIST instruction will restore the list/no list state to what it was before the last NLST instruction.

APSH - Push to stack. A stack is provided for assembly-time use by persons writing macros. This stack enables implementation by macros of structured constructs such as WHILE loops and IF-THEN-ELSE.

APOP - Pop one value from stack. The symbol in the label field is assigned the value. As with SET, the symbol is considered a temporary register and the value may be reassigned.

PEEK and POKE - See the section on PEEK and POKE.

SAMPLE PROGRAM "DISPLAY"

On the next page is a program which is included as an example. The source of this program is on the assembler disk as file DISPLAY/TXT.

When this program is assembled, loaded (with LOADM), and executed (with EXEC), it will ask for an address in RAM to be displayed. The 512-byte block of memory containing the address is then diplayed to the screen. The screen normally displays addresses in the range 0400 thru 05FF, so any of these addresses will have no effect. The up and down arrows will page through Ram, ENTER will allow entry of a new address, and BREAK will return to Basic. The function of this program is similar to the "P" command of the cassette version of CBUG.

The actual operation of this program may or may not be of interest to you. In any case, the source listing of it should, as it is an example of a number of the points mentioned thus far in this manual. It is also useful if you wish to learn how to set the display page of the video display, as you would in game programs.

```
0001 0E00                       NAM DISPLAY

                        *  THIS PROGRAM CAUSES ANY PAGE
                        *  IN RAM TO BE DISPLAYED ON
                        *  THE TEXT SCREEN.  IT CAN BE
                        *  USED FOR CHEATING AT
                        *  ADVENTURE GAMES.

0002 A1C1               POLCAT EQU $A1C1
0003 A282               OUTCHR EQU $A282
0004 A390               GETLIN EQU $A390
0005 A928               CLRSCR EQU $A928

0006 0E00 BDA928        START  JSR CLRSCR       CLEAR SCREEN
0007 0E03 8D17                 BSR OUTMSG       PRINT PROMPT
0008 0E05 8D2C                 BSR GETHEX       ADDRESS IN X
0009 0E07 8D72          DISPLA BSR SETSAM       SET DISPLAY
0010 0E09 BDA1C1        A@     JSR POLCAT       GET KEYSTROKE
0011 0E0C 27FB                 BEQ A@           LOOP TIL KEY
0012 0E0E 815E                 CMPA #'^         UP ARROW
0013 0E10 275D                 BEQ UPAROW       GO MOVE UP
0014 0E12 810A                 CMPA #$0A        DOWN ARROW
0015 0E14 275F                 BEQ DNAROW       GO MOVE DOWN
0016 0E16 8103                 CMPA #$03        BREAK
0017 0E18 2776                 BEQ BAKBAS       END OF PROGRAM
0018 0E1A 20E4                 BRA START        GO RE-PROMPT

0019 0E1C 308C0A        OUTMSG LEAX <MSG,PCR
0020 0E1F A680          D@     LDA ,X+          GET ONE LETTER
0021 0E21 260139               BEQ ?RTS         IF NULL, DONE
0022 0E24 BDA282               JSR OUTCHR       PRINT LETTER
0023 0E27 20F6                 BRA D@           LOOP FOR MESSAGE
0024 0E29 4144445245    MSG    FCC "ADDRESS? ",0

0025 0E33 BDA390        GETHEX JSR GETLIN       INPUT ONE LINE
0026 0E36 2556                 BCS BACKB2       IF BREAK, EXIT
0027 0E38 3001                 LEAX 1,X         POINT AT BUFFER
0028 0E3A 6FE2                 CLR ,-S          INITIALIZE ADDR
0029 0E3C 6FE2                 CLR ,-S          TO ZERO
0030 0E3E A680          GETDIG LDA ,X+          GET ONE LETTER
0031 0E40 272B                 BEQ ENDHEX       LEAVE IF END
0032 0E42 8030                 SUBA #'0         AT LEAST ZERO
0033 0E44 251C                 BLO REJ          REJECT IF NOT
0034 0E46 8109                 CMPA #9          NUMERIC?
0035 0E48 230A                 BLS OK           IF 0-9, FAT
0036 0E4A 8111                 CMPA #'A-'0      HEX LETTER?
0037 0E4C 2514                 BLO REJ          IF NOT LETTER, NG
0038 0E4E 8007                 SUBA #'A-'9-1    ADJUST
0039 0E50 810F                 CMPA #$0F        TOO BIG?
0040 0E52 220E                 BHI REJ          GO RE-PROMPT
0041 0E54 8D12          OK     BSR SH           SHIFT SUBTOTAL
0042 0E56 8D10                 BSR SH           FOUR BITS OVER
0043 0E58 8D0E                 BSR SH           TO MAKE ROOM FOR
```

```
0044 0E5A 8D0C          BSR SH          THE NEW DIGIT
0045 0E5C AB61          ADDA 1,S        ADD NEW DIGIT
0046 0E5E A761          STA 1,S         AND PUT IT BACK
0047 0E60 20DC          BRA GETDIG      GET NEXT DIGIT

0048 0E62 3262    REJ   LEAS 2,S        REMOVE OLD DATA
0049 0E64 8DB6          BSR OUTMSG      RE-PROMPT
0050 0E66 20CB          BRA GETHEX      AND TRY AGAIN

0051 0E68 6863    SH    ASL 3,S         SHIFT LOWER BYTE
0052 0E6A 6962          ROL 2,S         AND UPPER BYTE
0053 0E6C 39            RTS

0054 0E6D 3590    ENDHEX PULS X,PC      PULL RESULT

                  *
                  *  MOVE UP OR DOWN ONE PAGE
                  *
0055 0E6F 3089FE00 UPAROW LEAX -$200,X  LAST PAGE
0056 0E73 2092          BRA DISPLA      GO SHOW IT
0057 0E75 30890200 DNAROW LEAX $200,X   NEXT PAGE
0058 0E79 208C          BRA DISPLA      GO SHOW IT

                  *
                  *  SET DISPLAY ADDRESS
                  *
0059 0E7B 3410    SETSAM PSHS X         SAVE X
0060 0E7D CEFFC6        LDU #$FFC6      ADDRESS OF SAM
0061 0E80 8606          LDA #6          NUMBER OF BITS TO SEND
0062 0E82 1F89    C@    TFR A,B         COPY BIT NUMBER
0063 0E84 68E4          ASL 0,S         SHIFT NEXT BIT
0064 0E86 59            ROL B           MOVE IT INTO ADDRESS
                  *(IN THE NEXT STORE, DATA DOESN'T MATTER)
0065 0E87 A7C5          STA B,U         POKE THAT ADDRESS
0066 0E89 4A            DEC A           COUNT DOWN BITS
0067 0E8A 2AF6          BPL C@          LOOP FOR ALL BITS
0068 0E8C 3586          PULS D,PC       RESTORE AND RETURN

                  *
                  *  RETURN TO BASIC.
                  *  THIS SECTION PREVENTS SYNTAX
                  *  ERRORS IF THE EXEC WAS DONE
                  *  IN IMMEDIATE MODE.
                  *
0069 0E8E 3262    BACKB2 LEAS 2,S       POP RETURN ADDR
0070 0E90 9EA6    BAKBAS LDX $A6        BASIC EXECUTION POINTER
0071 0E92 8C03D4        CMPX #$03D4     IN KEYBOARD BUFFER
0072 0E95 22D5          BHI ?RTS        LEAVE IF NOT THERE
0073 0E97 6F80          CLR ,X+         CLEAR A BYTE
0074 0E99 6F80          CLR ,X+         AND ANOTHER
0075 0E9B 6F80          CLR ,X+         TO FLAG END OF LINE
0076 0E9D 39            RTS             BACK TO BASIC

0077 0E9E               END START

NO ERRORS FOUND
```

SECTION 2 - THE FUN PART

This section describes the interesting and unusual things which can be done with this assembler. You do not need to use any of the statements described here, and you can program perfectly well without them, but you will find them useful and will want to study and experiment with all of the topics in this section.

The features given in this section are in no particular order. Some are easy to learn about and very handy, such as local labels, while some are harder (like macros) and some are only used for certain special purposes (like checksums).

INCLUDE FILES

A text file on disk may be included as part of the current assembly by using the INCL pseudo-op. Some uses of this feature are:

Include a package of standard macros or subroutines;
Include a file of equates which are standard to several programs;
Include a series of component files into one master file;
Include files from another drive.

Example statements:

```
INCL FILE1
INCL 1:OTHERFIL/XYZ
```

If no drive is specified, the drive is assumed to be the last drive upon which an input file is opened (the main file or another include file). If no extension is specified, /TXT is assumed.

If the file name is incorrect or if such a file cannot be found on the disk, assembly will be aborted during pass 1.

Includes may be nested. The depth of nesting is only limited by the number of files currently allowed by Basic's FILES statement. If nesting is required, a FILES command should be typed in before running the assembler. If there are not enough files allocated, assembly will abort with an error message.

CONDITIONAL ASSEMBLY

Conditional assembly is a way of deciding whether or not certain statements should be assembled. It is an IF statement which is done at the time of assembly. It is useful in macros where different code is generated depending on what parameters are given, but it is also useful in programs which have no macros at all.

Programmers are commonly faced with the problem of having to maintain two or more versions of a program. Common examples include: (1) A stripped-down version versus a full-blown version; (2) Versions

which run on different kinds of computers; (3) Versions supporting different kinds of I/O. If different sources are kept, it becomes difficult to be sure that changes made to one version are made identically to all versions. The best solution to this problem is conditional assembly.

EXAMPLE:

```
*  VERSIONS FOR COLOR COMPUTER AND ANOTHER COMPUTER
COLOR  EQU -1          set not true for this assembly
       ...
       ...
       IFEQ COLOR
OUTPUT JMP $A30A       output routine for Color Computer
       ELSE
OUTPUT JMP $D286       output routine for other system
       ENDC
```

Thus lots of small changes may be made just by changing one line at the beginning of the program; and all differences are thoroughly documented.

The above discussion is not intended to imply that this is the only use for conditional assembly; it is a powerful tool which a programmer will find helpful in many situations.

The statements IFEQ, IFGE, IFGT, IFLE, IFLT, and IFNE are used to define the start of a conditional segment. The mnemonic is followed by a single expression which is evaluated in 16-bit arithmetic. If the condition specified in the mnemonic (=0, >=0, >0, <=0, <0, or <>0 respectively) is true, then assembly proceeds; otherwise no object code is generated (and no symbols defined) until the end of the conditional block. The condition may be reversed with the ELSE pseudo-op, which causes assembly either to restart or to be suspended. The conditional block is terminated with a ENDC statement.

IFs may be nested to a virtually unlimited depth. A sequence of IFxx-ENDC or IFxx-ELSE-ENDC will be ignored if it is within a block that is not being assembled; it will be processed normally if it is within a block that is being assembled.

Sample program BLKWHT gives another example of conditional assembly.

Something to think about:

```
IFNE *!.$FF
RMB 256-*!.$FF
ENDC
```

LOCAL LABELS

There are two kinds of local labels. There are general purpose local labels, and there are macro locals. The macro locals are discussed in the section on macros, and the general purpose local labels are discussed here.

Programs commonly get filled with many symbol names which have meaning only in the immediate vicinity of their definition. Examples include the labels used to branch over one or two statements or to form a small loop for setting or moving data. Over half of the symbols in a program may fall into this category.

Rather than trying to think of large numbers of unique names (which soon becomes difficult) or to clutter a symbol table with names like PUTIT, PUTIT2, PUTIT3, etc., the use of local labels allows the programmer to use short tags for this kind of symbol while leaving descriptive names free for more meaningful uses.

In this assembler, a local label is any single letter followed by an at-sign ("@"). It is defined only within a block bounded by blank lines. It is not listed in the symbol table. Each of the 26 local labels may be reused any number of times. Example:

```
MOVE    LDX #START
        LDY #WHERE
A@      LDA ,X+
        STA ,Y+
        CMPX #END
        BLO A@

        TST NULLIT
        BEQ A@
        CLR ,Y+
A@      RTS
```

The use of blank lines to delimit the scope of a symbol is not intended to cause the programmer to think of blank lines as a form of pseudo-op. Blank lines should be sprinkled around a program in any case; a readable program will have a blank line about every dozen lines. The logical division of a program into sections (such as the two sections in the subroutine above) which is defined by the blank lines corresponds exactly with the intended scope of the local labels. It is NOT intended (in spite of the example above) that local labels be reused in close proximity; if enough space is left between incarnations of local labels, the effect of blank lines on the labels need not concern the programmer.

SET LABELS

The SET pseudo-op is like EQU, only the value assigned to the symbol can be temporary. If the symbol appears later in another SET statement, then instead of saying "DEFINED TWICE" the assembler will just change the value. A label cannot be defined both with SET and

with some other method (such as EQU or as a statement label) since the latter definition is supposed to have effect throughout the entire program. Therefore, a symbol used with SET must be defined only with SETs and is called a "SET symbol" or "SET variable".

SET symbols may be thought of as variables, since, as with variables in a programming language, they are assigned values which have effect until they are reassigned. They may be used as counters (as in, COUNT SET COUNT-1) or for keeping track of such numbers as where the last byte of allocated RAM was. They are often used in macros, to save a value from one invocation of the macro to the next. They could be used in place of local labels, or even in place of normal labels, in order to save space in the symbol table and to speed up assembly. See the section on REORG for an example of using SET, and see sample program EX-SQR for another example.

MACROS

Macros are sections of source program which are to be assembled more than once. (That's not a very rigorous definition, but it will do for now.) Suppose, for example, that you want to shift the D register left, and are tired of writing ASL B / ROL A all over the program. Not only is it more work, and more to remember, but it's not very readable in the listing. It would be nice if you could just invent a new instruction, ASLD, which would generate those other two instructions. Well, you can! At the start of your program, you write:

```
ASLD    MACR
        ASL B
        ROL A
        ENDM
```

Now, whenever you write ASLD the assembler will go back and assemble those two statements.

The pseudo-op MACR signals the start of a macro definition. The name of the macro is "ASLD" as given in the label field. The next two lines are the "macro skeleton" and indicate what code should be generated when the macro is "invoked". The "ENDM" indicates the end of the definition.

Important: A macro is fundamentally different from a subroutine. In a subroutine, the code is generated only once and this is executed any number of times when the code is run. With a macro, the code is generated each time the macro is invoked. It is most useful where small sections of code are to be repeated many times.

Note: Although the macro name appears in the label field, it is not a label. It will not appear in the symbol table. It may not be used in an expression in an operand field. It is an instruction mnemonic, and may not be the same as an existing instruction or pseudo-op. It may, however, be the same as an existing macro; in this case the old macro definition is deleted.

The macro skeleton is not assembled until the macro is invoked. A syntax error will not be detected until this time. If an error is detected, the line will be listed; normally the skeleton is only listed when it is first defined. The option "M", however, will cause all of the macro expansions to be listed.

MACRO PARAMETERS

The usefulness of macros is greatly increased by the ability to pass parameters to the macro skeleton. Parameters are any text strings which are inserted in the text to be assembled. Example:

```
          NAM EXAMPLE2
ASL16     MACR
          ASL \0+1
          ROL \0
          ENDM
          . . .
          ASL16 VALUE
```

The symbol "\0" in the macro skeleton is replaced with the parameter when the macro is assembled. The text "VALUE" is the parameter which is given to the macro. The code which is assembled would read as:

```
          ASL VALUE+1
          ROL VALUE
```

Suppose we wanted a macro which could either do an ASR or an LSR on a 16 bit value. We could have two parameters, like this:

```
          NAM EXAMPLE3
SHIF16    MACR
          \1SR \0
          ROR \0+1
          ENDM
          . . .
          SHIF16 VALUE,A
          SHIF16 OTHER,L
```

This would generate this code:

```
          ASR VALUE
          ROR VALUE+1
          LSR OTHER
          ROR OTHER+1
```

The first parameter in a call is assigned the name \0, the second is \1, etc. After \9 comes \A, then \B, and so on through \Z. There can therefore be up to 36 parameters in a macro, which is far more than you will ever need. If a name is used in a macro but there were not that many parameters in the call, then that name is replaced by a null string (ie, no characters) in expanding the macro.

Parameters in a macro call are separated by commas, and the last parameter is followed by a space or end-of-line. If a parameter must contain a space or comma, then it must be surrounded by parentheses. A parameter may contain an end parenthesis as long as the parenthesis

is not followed by a blank or comma. Examples:

```
MACNAM XYZ,ABC,PDQ COMMENT
MACNAM (X,Y,Z),ABC,PDQ      First parameter = X,Y,Z
MACNAM XYZ,(A B,)C),PDQ     Second parameter = A B,)C
MACNAM ((XYZ)),ABC,PDQ      First parameter = (XYZ)
MACNAM XYZ,ABC,(P,),Q)      Fourth parameter = Q)
```

In the first example, the space after PDQ ends the parameters and COMMENT is ignored. In the second example, the first parameter is X,Y,Z and the parentheses were added because of the commas. The next example has a second parameter of A B,)C and the end parenthesis is not considered the end of the parameter because it is not followed by a comma or space (while the next end parenthesis is). In the next example, the parameter is (XYZ) and the other parentheses were added because the first set would be dropped. The last example will not generate the parameter P,),Q because of the comma after the end parenthesis. It will actually generate four parameters:
XYZ ABC P, Q)

LOCAL LABELS IN MACROS

Two kinds of local labels may be used within macros. The ordinary local labels (such as A@) may be used as usual, and macro locals may also be used (in a form such as \.AB). Examples of use of normal local labels with macros:

```
          NAM LOCALS
LOOP      MACR
A@        DEC B
          BNE A@
          ENDM
          . . .
          LDA #50
A@        LDB #100
          LOOP
          DEC A
          BNE A@
```

In this example, the first BNE will branch to the first A@ and the second BNE will branch to the second A@, even though the two loops will end up being nested. The scope of a local label defined within a macro skeleton will always be limited to that macro, even with no blank lines. The scope of a local label defined outside a macro may extend past a macro call but will not extend to within it.

MACRO LOCALS - ANOTHER KIND OF LOCAL LABELS

Macro locals are local labels which are for use only with macros. The scope of such a symbol is the macro skeleton within which it is defined. Unlike the other form of local labels, it is unaffected by blank lines. These symbols are generally used for labels which are to be defined within macros, since use of a normal symbol would result in a multiple-symbol-definition error when the macro was invoked a second time.

Macro local labels consist of a backslash, a period, and from one to four letters or numbers. If more than two letters or numbers are used, only the first two are used by the assembler. As with other symbols, a period, dollar sign, or underline (which is a back arrow) may also be used. The following is an example of use:

```
        NAM MACLOCS
DELAY   MACR
*  THIS DELAYS IN EITHER mSEC OR uSEC
\.C1    EQU (\0-3)/8
\.C2    EQU (\0-3)*125
        LDX #\.C\1 either C1 or C2
\.LOOP  LEAX -1,X
        BNE \.LOOP
        ENDM
        . . .
        DELAY 50,2  50 MILLISECONDS
        DELAY 400,1  400 MICROSECONDS
```

DATA GENERATION: Enhancements to FCB, etc.

Data is inserted in a program by means of the pseudo ops FCB, FDB, FCC, BSZ, and FCCS. FCB stores bytes, FDB stores 16 bit words, FCC stores ASCII code, BSZ stores zeroes, and FCCS stores screen code. Extra features have been added to FCB and FDB which are not normally found in 6809 assemblers, so that repetitive groups of data may be generated by one statement.

FCB - Form Constant Byte

This statement accepts an expression or list of expressions. Each expression must evaluate to an 8-bit value, or an error message is generated. The values may be either signed or unsigned. A 16-bit value (such as in the FDB statement) may be included by preceding an expression with ">". Examples:

```
010203      FCB 1,$02,100-97
FF          FCB -1
FF          FCB 255
010002      FCB 1,>2
```

A byte or group of bytes may be repeated. Examples:

```
010101      FCB 3[1]
01020102    FCB 2[1,2]
```

Repeats may be nested. The following line generates a linked list of empty entries, which could then be filled, sorted, relinked, etc. There are 20 entries in the list, and the last one has a zero link. (Remember that the assembler will only print out the first five bytes of the code generated.)

```
0E05FFFFFF FCB 19[>*+5,3[$FF]],>0,3[$FF]
```

FDB - Form Double Byte

The FDB statement is just like the FCB statement, only 16-bit (two byte) values are stored for each expression. Since all arithmetic in the assembler is done as 16-bit values, any expression can appear in an FDB. Just as ">" can be used to put a 16-bit value into an FCB statement, "<" can be used to put an 8-bit value into an FDB. The repeat-factor feature of FDB is identical to that of FCB. The linked list example above could have been written as:

```
0E05FFFFFF FDB 19[*+5,3[<$FF]],0,3[<$FF]
```

Exactly the same code is produced as with the FCB. If both the "<" and ">" were used, then it would be immaterial whether the mnemonic used was FCB or FDB.

FCC - Form Constant Character

This statement is used to insert ASCII code into the program. The usual form of the statement is:

```
4849        FCC "HI"
```

The 48 and 49 are the ASCII codes for H and I, which are generated by the assembler. The quotes delimit the string to be stored. Any symbol (except numbers) may be used as the delimiters. Examples:

```
4E455246    FCC /NERF/
2020202020 FCC .    .
```

If a comma appears after the end delimiter, the statement turns into an FCB statement. This is useful for adding carriage returns, nulls, etc., to a string. Example:

```
48490D00    FCC "HI",$0D,0
```

The statement: FCC "HI",0,"NERF" is not allowed, since "NERF" is not allowed in an FCB statement.

There is another form of the FCC statement, in which a count is given instead of delimiters. Examples:

```
4849        FCC 2,HI
202020      FCC 3,
4141        FCC 2,AAAAAAAAAAAAA
```

In the last example, only two A's are used and the rest treated as a comment.

FCCS - Form Constant Characters in Screen code

This statement is the same as FCC, except that the characters are stored using screen code. (See the chart of ASCII and Screen Code in this manual.) This code is used when text strings are to be moved directly into the memory used by the screen. Examples:

```
41427172    FCCS "AB12"
7131        FCCS "1",'1 (Think about it!)
```

BSZ - Block Store Zeroes

This statement takes a single expression, and generates that many zeroes. The syntax is the same as RMB, but RMB does not generate any data in the bytes it reserves, while BSZ does. The function of BSZ can also be performed by FCB. Example:

```
0000000000 BSZ 100
           RMB 100
0000000000 FCB 100[0]
```

EXTRA INSTRUCTIONS

Certain instructions have been added to this assembler which do not always appear in 6809 assemblers. These are:

CLRD - Clear D register. This generates CLR A; CLR B

TSTD - This generates STD -2,S which tests D for minus and zero

NEGD - This generates: COM A, COM B, SUBD #-1

RESET - This generates $3E which causes a processor reset and a jump to the restart address in ROM.

RHF - This generates $14 which puts the 6809 into a test mode wherein the address lines are continuously incremented. It is effectively a processor halt.

BRA # - The immediate mode is accepted on branch and conditional branch. BRA #3 will skip three bytes; BRN #$4F will skip over a CLRA instruction.

TFR # - Transfer and exchange will accept the immediate mode. This allows symbols to be used in place of register pairs.

PULS #- Push and pull S or U will accept the immediate mode. Symbols may therefore be used in place of register lists.

6800 opcodes - All 6800 source statements are accepted. For example, ABA ("add B to A") generates PSHS B / ADDA ,S+. See the section on 6800 cross assembly. Even if you're not into 6800 source code, the 6800 statements are sometimes easier to use (such as INX instead of LEAX 1,X) so it is worth looking these over.

SEF, SEIF, CLF, CLIF - these instructions set and clear the interrupt bits in the condition codes, and complement the 6800 instructions SEI and CLI (which don't effect the F bit).

CONDITIONAL RETURN

A conditional branch may be followed by the special symbol ?RTS. This causes the assembler to chose as the target of the branch the most recent RTS instruction, provided it is within 128 bytes. If there is no RTS within range, an RTS is inserted at the branch instruction and it becomes a three-byte instruction. Examples:

```
                    ...
1000 39             RTS
                    ...
1010 27EE           BEQ ?RTS
                    ...
2000 260139         BEQ ?RTS
```

THE "LIST" AND "NLST" PSEUDO-OPS

Sections of a program may be selectively removed from the listing. For example, such a section might be a large data table, or an include file, or a section of debugged code which is no longer of concern. The section is preceeded by the pseudo-op "NLST" and followed by "LIST". (Any line containing an error is listed anyway.)

Each "NLST" decrements a counter, and each "LIST" increments it again. Listing is performed whenever the counter is positive. This system allows flexibility of use. A "LIST" placed in front of the entire program will cause listing even of "NLST" areas, but still will not list areas which are "NLST" inside a "NLST". A "NLST" in front of the entire program will suppress listing, except for areas preceeded by "LIST" and followed by "NLST".

THE "END" PSEUDO-OP

The END statement is ordinarily considered to be just the end of a program, but there are a couple of other considerations.

A transfer address can be specified by putting an expression in the operand field. The value of this expression is saved in the object file and used as the address at which execution starts when EXEC is typed. This address is called the "transfer address", since execution is transfered to that address in order to run the program. If there are several END statements (such as with INCLUDE files) then the last transfer address encountered is used. When no transfer address exists, then the address of the start of the program is used.

An END statement is used to terminate input from a source file. If there is no END statement in a file, input continues until the actual end of the file is reached. It is a good idea to put END statements at the end of INCLUDE files to avoid getting extra blank lines in the listing. The assembly process will not terminate until the last input file has been read, regardless of END statements.

THE "REORG" PSEUDO OP

This statement resets the assembler's code generation pointer to the value it had just before the last ORG statement. It is used to continue assembly after some specification (such as of RAM) which required an ORG. In the following example, variables can be specified throughout a program by using SET variable RAMLOC to remember where the last ones have been allocated:

```
            ...
            ORG RAMLOC
VAR1        RMB 1
VAR2        RMB 10
VAR3        RMB 2
RAMLOC      SET *
            REORG
            ...
            ...
            ORG RAMLOC
VAR4        RMB 2
RAMLOC      SET *
            REORG
            ...
```

In this example, the ORG is used to specify a stack frame.

```
            ORG 0
LOCAL1      RMB 2
LOCAL2      RMB 2
LOCALS      EQU *
RETADR      RMB 2
PARAM1      RMB 2
PARAM2      RMB 1
            REORG
            LEAS -LOCALS,S
            LDD PARAM1,S
            STD LOCAL1,S
            ...
```

THE "PEEK" AND "POKE" PSEUDO-OPS

The operation of PEEK and POKE is similar to Basic's PEEK and POKE, but it is important to understand that the peeking and poking is done during assembly, not during the running of the program.

The PEEK statement reads the contents of the computer's memory during pass 1. The operand field is an expression which is used as an address. The content of that byte of memory is assigned as a permanent value to the symbol in the label field.

The POKE instruction alters the contents of the computer's memory during assembly time, during pass 1. The operand field contains two expressions separated by a comma. The first expression is an address into which the value of the second expression is placed.

The uses of PEEK and POKE are left to the imagination. Care should be exercised in the use of POKE, since its use could crash the assembler or cause subtle errors in its operation. PEEK and POKE are not found in many assemblers, and for good reason. Still, if you are a hardened hacker and know what you're doing, it may be useful.

CHECKSUMS

It is often a good idea to incorporate a self-test routine into a program, particularly when that program is going to be burned into a ROM (read only memory). If a few bits get changed in the program, it is better to have the program detect this and hang rather than continue with uncertain results.

The simplest way to do this is to use 16-bit adds to sum the entire program. The only problem is how to let the program know what the correct result should be. To facilitate this, this assembler supports a pseudo-op "CWORD" which generates a 16-bit constant such that the whole sum will be zero.

Since CWORD generates two bytes which are not code, it should not be placed where it would be executed. Normally it is placed at the end of a program as in the example below. If an odd number of bytes precedes the CWORD, an extra $FF is generated before the checkword so that the 16-bit adds will come out even.

If the entire program is not to be checksummed, the pseudo-op "CLRC" should be placed before the start of the summed area. This clears out the assembler's checksum and odd-even counter.

The assembler checksums all bytes generated, regardless of whether or not they are contiguous. In order to use this feature as described, the checksum should be generated only on contiguous areas of non-self-modifying code.

```
BEGIN . . .
      . . .
      LEAX BEGIN,PCR
      LDY #(LAST-BEGIN)/2
      CLRD
A@    ADDD ,X++
      LEAY -1,Y
      BNE A@
      SUBD #0   (or TSTD)
B@    BNE B@    hang here
      . . .
      . . .
      CWORD
LAST
      END BEGIN
```

LOCAL STACK

There is a stack which is reserved for the use of those writing macros. For example, a "FOR loop" macro could push values which a "NEXT" macro would pull off of the stack. To push some values, say

```
APSH VAL1,VAL2,VAL3
```

where VAL1 etc. are expressions which are evaluated (no forward references!) and pushed to the stack. Although several values can be pushed by one statement, they have to be pulled off again one at a time. Remember to pull them off in reverse order.

```
VAL3   APOP
VAL2   APOP
VAL1   APOP
```

The APOP statement sets a symbol to a temporary value as does SET. These symbols may not appear elsewhere in a label field except in another APOP or a SET.

In macros such as FOR, IF, WHILE, etc., it is a good idea to push a tag value last, so that the NEXT, ENDIF, or REPEAT can first pop that value and generate an error if the macros have not been nested correctly in the user's program.

The depth of the stack is 25 values. An error is generated if the stack overflows, and when the overflowed values are popped. The size of the stack could be changed by adding a line to AS/BAS such as

```
POKE &HE03+A,60: REM 60 BYTES ON STACK
```

Add this at a point after where AS%/BIN loads, and before it is run.

PROGRAMMING THE ASSEMBLER ITSELF

When we speak of assembler programming, we usually mean using an assembler to translate assembly language into machine language. With a powerful assembler, however, a lot of computation can be done at the time the assembler is running. This computational capability is intended to make it easier to write assembly language programs, but in this section we demonstrate the extent to which one can program the assembler. The assembler is an interpreter, and pseudo-ops are programming statements which are run by the interpreter.

Macros are the subroutines of this language, and conditional assembly gives us our IF statements. This assembler supports the ASK statement for input and the MSG statement for output. APOP and APSH give us a stack to play with. There is also a RPTM statement which gives us loops.

The example program, Towers, solves an ancient and well-known puzzle involving moving disks between three pegs. Programs which solve this puzzle are common homework assignments in programming classes and are

nothing new, but this particular program solves the puzzle while still in the assembler and never generates a byte of object code.

Another example program finds a square root by repeatedly trying division of trial answers. Of course, square roots can be more easily found in Basic, but one might envision a situation in which a source program would need to include numbers which might be most easily calculated within the assembler itself.

These two examples are extreme in that one would not generally wish to write stand-alone programs that do not generate code. The usefulness of the programmability of this assembler is in the flexibility it gives to writing complex assembly language programs. Whatever needs to be calculated in order to assemble the program can probably be calculated by instructions embedded right in the source code.

PAGE 0002 THE MICRO WORKS

```
0001 0E00                       NAM EX-SQR

                        *  THIS PROGRAM IS AN EXAMPLE OF
                        *  PROGRAMMING WITH MACROS.
                        *  IT IS NOT THE SORT OF THING
                        *  WHICH IS COMMONLY DONE WITH
                        *  MACROS, BUT IS AN INTERESTING
                        *  EXAMPLE.

                        *  IT FINDS THE INTEGER SQUARE
                        *  ROOT OF A NUMBER ITERATIVELY.

0002 0E00               SQ      MACR
0003 0E00                       IFGT CNT
0004 0E00               QUOT    SET NUM/TRY
0005 0E00               TRY     SET (TRY+QUOT)/2
0006 0E00               CNT     SET CNT-1
0007 0E00                       SQ
0008 0E00                       SQ
0009 0E00                       ENDC
0010 0E00                       ENDM

0011 00A9               NUM     ASK "SQUARE ROOT OF WHAT?"
0012 0010               CNT     SET 16
0013 0054               TRY     SET NUM/2
0014 0E00                       SQ
0015 0E00                       MSG "ANSWER IS ",TRY

                        *  THIS STORES THE ANSWER IN
                        *  MEMORY.  DO NOT TRY TO EXEC
                        *  THE OBJECT OF THIS PROGRAM,
                        *  AS THIS IS ALL THAT IS THERE.
0016 0E00 000D                  FDB TRY

0017 0E02                       END
```

```
0001 0E00                       NAM TOWERS

                         *  THIS IS AN UNUSUAL EXAMPLE OF PROGRAMMING
                         *  WITH MACROS.

                         *  THE SOLUTION OF THE "TOWERS" PROBLEM IS
                         *  PRINTED OUT DURING ASSEMBLY, AND NO OBJECT
                         *  CODE IS GENERATED.

                         *  THE PUZZLE IS THIS:  THERE ARE THREE POLES,
                         *  AND WE WISH TO MOVE A STACK OF DISKS FROM
                         *  POLE 1 TO POLE 2.  EVERY DISK IS SMALLER
                         *  THAN THE DISK BELOW IT, AND NO DISK MAY BE
                         *  PUT ON TOP OF A SMALLER ONE.  ONLY ONE DISK
                         *  AT A TIME MAY BE MOVED.

0002 0E00                MOVE    MACR
0003 0E00                        IFNE \3
0004 0E00                        MOVE \0,\2,\1,\3-1
0005 0E00                        MSG "MOVE FROM \0 TO \1"
0006 0E00                        MOVE \2,\1,\0,\3-1
0007 0E00                        ENDC
0008 0E00                        ENDM

0009 0004                HMANY   ASK "MOVE HOW MANY? "
0010 0E00                        MOVE POLE1,POLE2,SPARE,HMANY
0011 0E00                        END

NO ERRORS FOUND
```

```
MOVE FROM POLE1 TO SPARE
MOVE FROM POLE1 TO POLE2
MOVE FROM SPARE TO POLE2
MOVE FROM POLE1 TO SPARE
MOVE FROM POLE2 TO POLE1
MOVE FROM POLE2 TO SPARE      (THE ANSWER IS
MOVE FROM POLE1 TO SPARE      ACTUALLY PRINTED
MOVE FROM POLE1 TO POLE2      OUT FIRST,
MOVE FROM SPARE TO POLE2      DURING PASS 1.)
MOVE FROM SPARE TO POLE1
MOVE FROM POLE2 TO POLE1
MOVE FROM SPARE TO POLE2
MOVE FROM POLE1 TO SPARE
MOVE FROM POLE1 TO POLE2
MOVE FROM SPARE TO POLE2
```

```
0001 0E00                          NAM BLKWHT

                           ;       THIS IS AN EXAMPLE OF BOTH THE ASK STATEMENT
                           ;       AND CONDITIONAL ASSEMBLY.

                           ;       IT SETS THE SCREEN TO EITHER BLACK, WHITE,
                           ;       OR RED, DEPENDING ON WHICH COLOR IS TYPED
                           ;       WHEN IT IS BEING ASSEMBLED.

                           ;       LOCATION $8A IS ALWAYS ZERO, SO WE'LL USE IT
                           ;       INSTEAD OF SAYING LDX #0:
0002 008A                  ZERO    EQU $8A

                           ;       THESE NUMBERS ARE ARBITRARY:
0003 0001                  BLACK   EQU 1
0004 0002                  WHITE   EQU 2
0005 0003                  RED     EQU 3

                           ;       THIS IS THE OPERATOR INPUT:
0006 0002                  COLOR   ASK "WHAT COLOR? "

                           ;       THIS IS THE CONDITIONAL ASSEMBLY.  THE
                           ;       MINUS SIGN CHECKS FOR EQUALITY:
0007 0E00                          IFEQ COLOR-BLACK
0008 0E00                          LDA #$80
0009 0E00                          ELSE
0010 0E00                          IFEQ COLOR-WHITE
0011 0E00 86CF                     LDA #$CF
0012 0E02                          ELSE
0013 0E02                          IFEQ            COLOR-RED
0014 0E02                          LDA #$BF
0015 0E02                          ELSE
0016 0E02                          MSG             "THAT WASN'T RIGHT"
0017 0E02                          FAIL            "COLOR NO GOOD"
0018 0E02                          ENDC
0019 0E02                          ENDC
0020 0E02                          ENDC

                           ;       THE A REGISTER NOW CONTAINS THE
                           ;       COLOR CODE.  THIS SETS THE SCREEN:
0021 0E02 8E0400                   LDX #$400
0022 0E05 A780             A@      STA ,X+
0023 0E07 8C0600                   CMPX #$600
0024 0E0A 25F9                     BLO A@

                           ;       THIS PAUSES WHILE WE ADMIRE THE SCREEN:
0025 0E0C 9E8A                     LDX ZERO
0026 0E0E 3D               B@      MUL             (TIME DELAY ONLY)
0027 0E0F 3001                     INX
0028 0E11 26FB                     BNE B@

                           ;       END OF PROGRAM
0029 0E13 39                       RTS
0030 0E14                          END

NO ERRORS FOUND
```

```
0001 0E00                       NAM TEXTOUT

                        *  THIS IS AN EXAMPLE OF THE USE OF MACROS.

                        *  THE MACRO "TEXT" IS USED WHENEVER
                        *  TEXT IS TO BE PRINTED ON THE SCREEN.
                        *  IT GENERATES THE APPROPRIATE CALL TO
                        *  THE SUBROUTINE "PRINT".

                        *  THERE ARE EASIER WAYS TO DO THIS
                        *  PARTICULAR JOB, BUT THIS SHOWS A USE
                        *  OF MACROS WHICH IS TYPICAL IN MANY WAYS.

0002 0E00               TEXT    MACR
0003 0E00                       PSHS X
0004 0E00                       LEAX \.TX,PCR MAKE THIS P.I.C.
0005 0E00                       LBSR PRINT
0006 0E00                       PULS X
0007 0E00                       BRA \.OV SKIP OVER TEXT
0008 0E00               \.TX    FCC "\0"
0009 0E00                       IFC \1,!
0010 0E00                       FCB $0D GO TO NEW LINE
0011 0E00                       ELSE
0012 0E00                       FCB $20 OTHERWISE A SPACE
0013 0E00                       ENDC
0014 0E00                       FCB 0 END STRING WITH A NULL
0015 0E00               \.OV
0016 0E00                       ENDM

                        *  NOW, FOR THE MAIN PROGRAM.  IT DOESN'T DO
                        *  MUCH, BUT ILLUSTRATES THE USE OF THE MACRO.

                        *  THIS PRINTS THE WORD, FOLLOWED BY A SPACE:
0017 0E00                       TEXT TESTING
0018 0E16                       TEXT ONE
0019 0E28                       TEXT TWO
                        *  THIS PRINTS THE WORD, FOLLOWED BY A RETURN:
0020 0E3A                       TEXT THREE,!
                        *  THE PARENTHESES ARE NEEDED, BECAUSE OF
                        *  THE COMMA AND SPACE.  THEY ARE NOT PRINTED.
0021 0E4E                       TEXT (TESTING, ONE)
0022 0E69                       TEXT (TWO THREE),!

0023 0E81 39                    RTS

                        *  THIS IS THE SUBROUTINE WHICH
                        *  WE JUST CALLED SIX TIMES:

0024 0E82 A680          PRINT   LDA ,X+
0025 0E84 27FB                  BEQ ?RTS
0026 0E86 AD9FA002              JSR [$A002]     OUTPUT CHARACTER
0027 0E8A 20F6                  BRA PRINT

0028 0E8C                       END

NO ERRORS FOUND
```

SECTION 3 - DETAILS AND DULL STUFF

In this section we give details about certain parts of the assembler and about certain aspects of 6809 assembly language. Such subjects as expression evaluation or permissable symbol names are normally learned by example and experiment, but a summary of the rules will probably point out features of which you would not otherwise be aware. Topics such as branch conditions or error messages are included because it is handy to have them summarized somewhere.

EXPRESSIONS AND CONSTANTS

A general expression processor is used to evaluate any portion of an operand field where an arithmetic value is needed. Thus anywhere a symbol may be used as an operand or part of an operand, one may use an expression of arbitrary complexity. The following unusual examples illustrate some capabilities of the expression processor:

```
LDX #3-(XYZ*$22+'&!R3)
LDA [1-(2-(3-(4-(5-6)))),X]
FDB 1,$2,%11,@4,5H,110B,7Q,'8-'0
```

Note that spaces are not allowed in an expression, as they are used as field terminators. Commas are not allowed in expressions; they separate expressions (as in the FDB instruction in the above example) or can signal indexed mode (as in the LDA instruction).

Parentheses may be used at will to specify the grouping of the operations. (Brackets may not be used as these are used for indirect addressing.) In the absence of parentheses, the order is (1) unary minus, (2) all operations except plus and minus, in order of left to right, and (3) plus and minus last. This is a standard order of evaluation such as one might expect.

All operations are performed in 16 bit arithmetic and no overflow or carry conditions are recognized, so operations may be treated as either signed or unsigned.

In addition to +, -, *, and /, the logical operations AND (!.), OR (!+), and EXCLUSIVE OR (!X) are included, with the same precedence as * and /. Exponentiation is also included, specified by an up arrow (^), such that 3^4 is the same as 3*3*3*3. Also included are shift left, shift right, rotate left, and rotate right (!<, !>, !L, and !R respectively), which shift the left operand the number of bits specified in the right operand. Examples:

```
5!<2   is  20
5!>2   is  1
5!R2   is  16385
```

Note that 2^N is the same as 1!<N.

Constants may be specified in base 2, 8, 10, or 16. Default is base

10. A constant may be preceded by a %, @, &, or $ to indicate bases 2, 8, 10, or 16 repectively. Alternatively a suffix of B, Q (not O), or H may be used for bases 2, 8, or 16; there is no suffix for base 10. A constant may not start with a letter or it will be interpreted as a symbol; therefore FFH must be written as 0FFH to comply with this rule. Better yet, write $FF; the prefix form is more widely used as it is more readable.

ASCII constants are preceded by a single quote. There is no end quote required, though it is allowed. Two characters may be included between quotes as long as the second one is a letter or number.

The symbol "*" may be used to represent the value of the assembler's program counter. For example, the statement FDB LABEL-* will store the relative address of LABEL.

SUMMARY OF OPERATIONS

	Example	Result in hex
addition	1+1	0002
subtraction	2-1	0001
multiplication	2*2	0004
division	7/2	0003
exponentiation	2^5	0020
logical AND	$1234!.$FF	0034
logical OR	$1234!.$FF	12FF
exclusive OR	$1234!X$FF	12CB
shift left	$10!<2	0040
shift right	$10!>2	0004
rotate left	$10!L15	0008
rotate right	$10!R15	0020

SUMMARY OF FACTORS

SYMBOL	symbol
1234	decimal number
&1234	decimal number
1234H	hex number
$1234	hex number
1234Q	octal number
@1234	octal number
1010B	binary number
%1010	binary number
'A	character
'A'	character
'AB'	two characters (second must be letter or number)
(EXP)	any valid expression inside parentheses
*	current value of program counter
-FACTOR	unary minus with any of the above
+FACTOR	unary plus with any of the above

SYMBOLS

A symbol is a letter or string of letters, numbers, and symbols, which is used to represent a numeric value such as an address in memory. The first character of a symbol name must be a letter. The rest of the characters may be letters, numbers, or the symbols dollar sign, period, or underline. (The underline character displays as a back arrow on the Color Computer, and is obtained by pressing shift up arrow.)

Symbol names are limited to six characters. More than six characters may be used, but the remaining characters will be ignored. If you wish to have such names, use the pseudo-op "LONGVR" to suppress the error message normally generated by names longer than six characters.

A symbol normally has one value throughout the entire program. Its value is set either by using the symbol as a label (putting the name to the left of an instruction) or by using the EQU statement. If you wish to give a symbol a series of different values, use the SET pseudo-op; this is like EQU only the value can be changed by another SET. See the section called "SET LABELS".

There are also two kinds of "local labels". These are discussed in the sections called "LOCAL LABELS" and "MACRO LOCALS".

Some types of statements require that the symbols which they reference are previously defined. An example is the RMB statement. The statement RMB XYZ reserves some memory bytes, and it must be known during the first pass how many to reserve. Therefore XYZ must have been defined in a line previous to the RMB statement. The statement XYZ RMB $100-XYZ will not work, since XYZ is not considered defined until after the line in which it appears. These constraints are the same as those of most two-pass assemblers.

There are no reserved symbol names, but it is not a good idea to use the names A, B, D, X, Y, S, U, PC, and PCR as symbols. A statement such as LDD A,X will assume that the A is a register, not a symbol. The statement LDD X will assume that X is a symbol; other assemblers would treat X as a register but correct syntax is to require a comma before the X in indexed addressing.

6809 INSTRUCTION SET

ABX	Add B to X (Unsigned)
ADCA or ADCB mem	Add memory and carry to accumulator
ADDA or ADDB mem	Add memory to accumulator
ADDD mem	Add memory(2) to D register (A:B)
ANDA or ANDB mem	Logical And memory to accumulator
ANDC #value	Same as ANDCC
ANDCC #value	Logical And Immediate with condition codes
ASLA or ASLB	Arithmetic shift left accumulator
ASL mem	Arithmetic shift left memory
ASRA or ASRB	Arithmetic shift right accumulator
ASR mem	Arithmetic shift right memory

BCC label	Branch if carry clear
BCS label	Branch if carry set
BEQ label	Branch if equal set
BGE label	Branch if greater or equal (signed)
BGT label	Branch if greater than (signed)
BHI label	Branch if higher (unsigned)
BHS label	Branch if higher or same (unsigned)
BITA or BITB mem	Bit test with memory (And but save reg.)
BLE label	Branch if less than or equal (signed)
BLO label	Branch if lower (unsigned)
BLS label	Branch if lower or same (unsigned)
BLT label	Branch if less than (signed)
BMI label	Branch if minus (N set)
BNE label	Branch if not equal
BPL label	Branch if plus (N not set)
BRA label	Branch always
BRN label	Branch never
BSR label	Branch to subroutine
BVC label	Branch if overflow clear
BVS label	Branch if overflow set
CLRA or CLRB	Clear accumulator to zero
CLR mem	Clear memory byte to zero
CMPA or CMPB mem	Compare memory to accumulator
CMPD mem	Compare D register (A:B) to memory(2)
CMPS, CMPU, CMPX, CMPY mem	Compare index register to memory(2)
COMA or COMB	One's complement (bit flip) accumulator
COM mem	One's complement memory
CWAI #value	And with CC reg and wait for interrupt
DAA	Decimal Add Adjust accumulator A
DECA or DECB	Decrement accumulator (note carry not set)
DEC mem	Decrement memory (note carry not set)
EORA or EORB mem	Exclusive Or
EXG reg, reg	Exchange two registers
INCA or INCB	Increment accumulator (note carry not set)
INC mem	Increment memory (note carry not set)
JMP mem	Jump to address
JSR mem	Push PC and jump to address
LDA or LDB mem	Load accumulator
LDAA or LDAB mem	Same as LDA and LDB
LDD mem	Load D register (A:B) with memory(2)
LDX, LDY, LDU, LDS mem	Load index register with memory(2)
LEAX, LEAY mem	Load with effective address (sets Z bit)
LEAU, LEAS mem	Load with effective address (Z not set)
LSLA or LSLB	Logical shift left (same as ASL)
LSL mem	Logical shift left (same as ASL)
LSRA or LSRB	Logical shift right (zero fill)
LSR mem	Logical shift right (zero fill)
MUL	Multiply A x B, result in D
NEGA or NEGB	Negate (2's complement)
NEG mem	Negate memory byte
NOP	No operation
ORA or ORB mem	Inclusive Or
ORAA or ORBB mem	Same as ORA and ORB
ORCC #value	Or to condition codes register
PSH reg,reg,...	Same as PSHS

PSHS reg,reg,...	Push registers to system stack
PSHU reg,reg,...	Push registers, using U as stack pointer
PUL reg,reg,...	Same as PULS
PULS reg,reg,...	Pull registers from system stack
PULU reg,reg,...	Pull registers, using U as stack pointer
ROLA or ROLB	Rotate left through carry
ROL mem	Rotate left through carry
RORA or RORB	Rotate right through carry
ROR mem	Rotate right through carry
RTI	Return from interrupt
RTS	Return from subroutine (pull PC)
SBCA or SBCB mem	Subtract, and subtract carry
STA or STB mem	Store accumulator
STAA or STAB mem	Same as STA and STB
STD mem	Store D register (A:B) to memory(2)
STX, STY, STU, STS mem	Store index register to memory(2)
SUBA or SUBB mem	Subtract memory from accumulator
SUBD mem	Subtract memory(2) from D register (A:B)
SWI, SWI2, SWI3	Software interrupt
SYNC	Halt until interrupt (masked or not)
TFR reg,reg	Transfer reg to reg
TSTA or TSTB	Test for zero and negative
TST mem	Test for zero and negative

ADDRESSING MODES

In the list of instructions above, the word "mem" in the left column indicates that a memory address should be specified with the instruction. These memory addresses should be in the one of the following forms:

#expression	immediate addressing mode - the value given is used as an operand. This mode is required for instructions such as ANDCC, ORCC, and CWAI. It is not allowed for memory-modifying instructions such as stores, shifts, etc.
expression	extended or direct mode is used by the assembler - the value given is used as an absolute address.
[expression]	indirect - the value given is used as the address of the address of the operand.
expression,index	indexed - where "index" is one of the registers X, Y, U, S, or PC.
,index	indexed - same as 0,index.
,index+	auto increment - the index register X, Y, U, or S is incremented after use.

,index++	auto increment - the index register is incremented twice after use.
,-index	auto decrement - the index register X, Y, U, or S is decremented before use.
,--index	auto decrement - the index register is decremented twice before use.
acc,index	accumulator offset - "acc" is A, B, or D and is used as a signed offset to X, Y, U, or S.
expression,PCR	PC relative - the address specified by the value of the expression is referenced by means of the PC indexed mode.
[expression,index] [,index] [,index++] [,--index] [acc,index] [expression,PCR]	indirect indexed modes

SHORT VERSUS LONG ADDRESSING

There are many cases where an instruction can be assembled in either a long or a short form. Examples:

```
LDA XYZ       if XYZ < $100, then direct addressing may be used
STA PDQ,X     this may take 2, 3, or 4 bytes, depending on PDQ
LDX [ABC,Y]   this may take 3 or 4 bytes
```

The assembler must be able to decide on the length of the instruction during the first pass, so that all subsequent symbols may be defined correctly. If all symbols used in such an instruction have already been defined, then the shortest possible addressing mode may be used. It is therefore a good idea to define variables at the top of the program instead of at the bottom. Where there is an undefined symbol in an expression, the longest form of the instruction will be used by default, since that will work regardless of the values of the undefined variables.

If the programmer knows which mode should be used, the symbols "<" and ">" may be used to specify it. Examples:

```
LDA <XYZ      use direct addressing
STA <PDQ,X    use 8 bit offset (not 5 bit unless PDQ defined)
LDX >[ABC,Y] use 16 bit offset
```

The use of direct addressing is dependent not on the expression being less than $100 (as we sometimes imply) but in the high byte being equal to the assumed value of the DP register. This is controlled by the SETDP pseudo-op as described in the next section.

USE OF THE DIRECT PAGE REGISTER

The 6809 processor contains an 8-bit register called the "Direct Page Register". The only use of this register is to hold the upper byte of addresses for the direct addressing mode. It normally contains zero.

An instruction such as LDA $1234 will take three bytes of memory and take five microseconds to execute. An instruction such as LDA $12 (using the direct addressing mode) will take two bytes of memory and take four microseconds to execute. It is therefore a good idea to use the direct addressing mode for the bulk of a program's variables.

The Basic interpreter in the Color Computer keeps the DP (Direct Page register) set to zero; therefore the 256 bytes of RAM in the range $0000 through $00FF may be accessed in direct mode. When calling a subroutine in the ROM it is necessary that DP be zero; the subroutine may crash if it is not. While running another machine language program, however, the DP may be set to the upper byte of the address of the page where the program's variables are stored.

In order to use direct addressing, it is necessary both to tell the assembler to create direct-addressing instructions, and to insert the instructions which set the DP. The former is done with the "SETDP" pseudo-op; it sets the range of addresses which will be assumed to work with direct addressing. The latter is done with TFR A,DP or TFR B,DP or PULS DP. Example:

```
RAM     EQU $1200
        LDA #RAM/256
        TFR A,DP
        SETDP RAM/256
        LDA #$1234      direct
        LDA #$12        not direct
        . . .
        CLR A
        TFR A,DP
        SETDP 0
        LDA #$1234      not direct
        LDA #$12        direct
```

ERROR MESSAGES

There are three kinds of error messages which the assembler can generate. These are fatal errors, program errors, and optional errors.

Fatal errors are printed to the screen regardless of the listing unit. They are followed by an immediate termination of the assembly run. They can be caused by any of the following:

-- A file name which is syntactically incorrect
-- An input file which cannot be found
-- A "phasing error", which indicates that some other error has caused a mismatch between pass 1 and pass 2 symbol values
-- A memory overflow

-- Incorrect format of the parameter string passed from Basic
-- A macro definition which never terminates

A memory overflow error may be caused by too many symbols, macro nesting so deep as to cause a stack overflow, or a macro which calls itself infinitely.

Program errors (such as syntax errors) are printed to the listing unit regardless of whether or not a listing is being generated. If option "E" is in effect the listing will go into single-step mode. The errors are as follows:

BAD MNEMONIC - Mnemonic misspelled, macro name undefined, or label is in mnemonic field.
SYNTAX ERROR - The format of the operand field is not correct.
BAD LABEL - A character was found in the label field that is not allowed in a label.
DEFINED TWICE - A symbol has appeared twice in the label field, other than in a SET instruction.
UNDEFINED - A reference is made to a symbol which is not given a value anywhere in the program.
BYTE OVERFLOW - A number was expected in the range -128 thru +255.
NOT IMMEDIATE - This instruction (such as ORCC) requires that the operand be in immediate mode.
BRANCH TOO FAR - The target of the branch is more than 128 bytes away. It should be changed to a long branch.
FORWARD REF - This instruction (such as ORG or RMB) does not allow a forward reference in its operand, in order that the symbol table may be completely built during pass 1.
TERMINATOR - The instruction is syntactically wrong. It was right up to a point, but then there were additional characters which must be ignored in order to assemble the statement.
TRUNCATED - A symbol name of more than six characters has been truncated to six characters.
USER STACK OVF - The local stack has overflowed (see that section).

As each error line is listed, the line number of the last error is given. At the end of the listing, the line number of the last error is given. This enables one to quickly find all error lines without searching through the entire listing.

The optional errors are treated like the program errors above, but are only generated if requested. They are:

ZERO BYTE - A byte of object code is equal to zero. This message is enabled by the "Z" option or by the NOZER pseudo-op. It is useful for assembling programs to be embedded within Basic program lines.
COULD BE SHORT - A long branch was used where a short branch would work. This is enabled by option "W". It is useful for making a debugged program shorter for its final version.
USER - The "FAIL" pseudo-op has been assembled. The operand field of the instruction is also listed. This instruction is used in conjunction with conditional assembly within macros to flag conditions caused by illegal parameters.

BRANCH CONDITIONS

There are 16 different branch instructions on the 6809. Each of these may be either a short or long branch. A short branch has a range of plus or minus 127 bytes; a long branch may branch anywhere in memory. A long branch is specified by prefixing an "L" to the mnemonic, such as LBEQ instead of BEQ. These are the 16 branches:

20 BRA This will always branch. The instruction LBRA is similar to JMP except that relative addressing is used.
21 BRN Branch Never. About the only use for this instruction is to skip over its own operand, and thus be a one-byte skip.
22 BHI Branch if Higher. C=0 and Z=0. For unsigned comparison.
23 BLS Branch if Lower or Same. C=1 or Z=1. For unsigned comparison.
24 BCC Branch if Carry Clear. C=0. Indicates no overflow from an unsigned add. Same as BHS.
24 BHS Branch if Higher or Same. C=0. For unsigned comparison. Same as BCC.
25 BCS Branch if Carry Set. C=1. Indicates overflow from an unsigned add. Same as BLO.
25 BLO Branch if lower. C=1. For unsigned comparison. Same as BCS.
26 BNE Branch if not equal. Z=0. For signed or unsigned comparison, or nonzero result from ADD, AND, BIT, load, and store.
27 BEQ Branch if equal. Z=1. For signed comparison, or indicating a zero result from ADD, AND, BIT, load, and store.
28 BVC Branch if overflow clear. V=0. For no overflow on signed operations.
29 BVS Branch if overflow set. V=1. For overflow on signed operations.
2A BPL Branch if plus. N=0. Indicates positive result from ADD, SUB, AND, BIT, load, store, etc.
2B BMI Branch if minus. N=1. Indicates negative result from ADD, SUB, AND, BIT, load, store, etc.
2C BGE Branch if Greater or Equal. N xor V = 0. For signed comparison.
2D BLT Branch if Less Than. N xor V = 1. For signed comparison. (BGE & BLT are the same as BPL & BMI after a load or store.)
2E BGT Branch if Greater Than. Z or (N xor V) = 0. For signed comparison, or loads and stores of signed numbers.
2F BLE Branch if Less Than or Equal. Z or (N xor V) = 1. For signed comparison, or loads and stores of signed numbers.

ASCII CODE AND SCREEN CODE

Listed below is the ASCII code - the American Standard Code for Information Interchange. This is the code used by assembly language to represent text strings, and is the code which the assembler expects its source code to be in. Also listed below is the Color Computer screen code, which is used when storing directly to the screen. The FCC pseudo-op generates ASCII code, and FCCS generates screen code.

ASCII	SCREEN	CHAR	ASCII	SCREEN	CHAR	ASCII	SCREEN	CHAR
20	60	space	40	40	@	60	--	`
21	61	!	41	41	A	61	01	a
22	62	"	42	42	B	62	02	b
23	63	#	43	43	C	63	03	c
24	64	$	44	44	D	64	04	d
25	65	%	45	45	E	65	05	e
26	66	&	46	46	F	66	06	f
27	67	'	47	47	G	67	07	g
28	68	(	48	48	H	68	08	h
29	69	)	49	49	I	69	09	i
2A	6A	*	4A	4A	J	6A	0A	j
2B	6B	+	4B	4B	K	6B	0B	k
2C	6C	,	4C	4C	L	6C	0C	l
2D	6D	-	4D	4D	M	6D	0D	m
2E	6E	.	4E	4E	N	6E	0E	n
2F	6F	/	4F	4F	O	6F	0F	o
30	70	0	50	50	P	70	10	p
31	71	1	51	51	Q	71	11	q
32	72	2	52	52	R	72	12	r
33	73	3	53	53	S	73	13	s
34	74	4	54	54	T	74	14	t
35	75	5	55	55	U	75	15	u
36	76	6	56	56	V	76	16	v
37	77	7	57	57	W	77	17	w
38	78	8	58	58	X	78	18	x
39	79	9	59	59	Y	79	19	y
3A	7A	:	5A	5A	Z	7A	1A	z
3B	7B	;	5B	5B	[	7B	--	{
3C	7C	<	5C	5C	\	7C	--	|
3D	7D	=	5D	5D	]	7D	--	}
3E	7E	>	5E	5E	^	7E	--	~
3F	7F	?	5F	5F	_ or ←	7F	--	rub

Note that on the screen of the Color Computer, the character "^" appears as an up-arrow, the character "_" appears as a back-arrow, and the lowercase letters appear as uppercase letters, only inverted. Also note that carriage return (ENTER key) is $0D in ASCII.

SECTION 4 - PROGRAMMING IN 6809 ASSEMBLY LANGUAGE

This section contains information on a number of topics related to programming in 6809 assembly language and to the Color Computer in particular. The first section is for beginners; if you already know 6809 then you will want to skip that section but you will definitely want to read the rest of this chapter to make the most of your Color Computer.

BEGINNERS GUIDE TO 6809 ASSEMBLY LANGUAGE PROGRAMMING

Assembly language can't be taught in a few paragraphs or even by one whole book. The basic concepts, like what a register is or what bytes are for, can't be covered here. This section assumes a basic familiarity with microprocessors but gives you a lesson in translating theory into practice on the 6809.

Here is the first sample program. This could be entered using a text editor, assembled using the assembler, loaded using LOADM, and run by typing EXEC. It adds two plus two.

```
        LDA #2
        ADDA #2
        STA ANSWER
        RTS
ANSWER  RMB 1
        END
```

This uses the A register, which is a one-byte general-purpose register. See page 56 for a diagram of the registers.

The LDA #2 puts a two into the A register. The ADDA #2 adds two to this, leaving four in the A register. The STA ANSWER will store the contents of the A register into the memory byte which we will name ANSWER. The RTS will stop the program; it returns control to Basic after we EXEC the program. The word ANSWER in the next line is a label; it starts in the first column and tells the assembler that this is the address which we were referring to when we used the symbol ANSWER above. The RMB 1 means "Reserve one Memory Byte" for storing the answer. The END tells the assembler that this is the end of the source file.

The number signs ("#") in the first two lines of the program mean that the two is an actual number to use; if we just said LDA 2 then the instruction would mean "Load A from memory location 2". Using the number sign means that the instruction is in "immediate mode".

Using the A register as we have only allows us to work with numbers up to 255, since this is all that can fit in one byte. The A register is actually half of the D register, and we can use the whole D register if we want to go up to 65535. (Refer again to the "Programmer's Model" on page 56.) Here is the new version of our program using two-byte numbers:

```
        LDD #1234
        ADDD #1357
        STD ANSWER
        RTS
ANSWER  RMB 2
        END
```

Note that now we have to reserve two bytes for ANSWER by saying RMB 2.

Of course, after adding the two numbers we just stuck the answer in memory somewhere and left it there; it doesn't print the result on the screen. We are not going to go into the techniques of printing numbers to the screen, but we will print letters. This program prints the word "HI":

```
        LDA #'H
        JSR $A282
        LDA #'I
        JSR $A282
        RTS
        END
```

The expression 'H means "the number which is the code for H". This assembler (and the whole Color Computer) uses the ASCII code, in which A is 65, B is 66, and so on. The JSR means "jump to subroutine" and the $A282 is the address of a subroutine in the Color Computer's permanent memory which displays characters to the screen.

Actually, there is a more direct way of writing to the screen. The screen is just a section of memory, so we can store characters into it and they will appear. The screen goes from $0400 through $05FF. This program puts the word "HI" onto the screen about in the middle.

```
        LDA #'H
        STA $510
        LDA #'I
        STA $511
        RTS
        END
```

The screen doesn't quite use standard ASCII code, since it can't display lowercase but can display inverted characters and graphics symbols. Therefore, when storing directly to the screen, refer to the chart given in this manual for ASCII and Screen Code.

The following program uses indexed addressing to cover the entire screen with question marks. Of course, as soon as the program returns to Basic the screen will scroll, so the bottom two lines will no longer contain question marks, but we will know that they all were there.

```
        LDX #0
LOOP    LDA #'?
        STA $400,X
        LEAX 1,X
        CMPX #$200
        BLO LOOP
        RTS
        END
```

The statement LEAX 1,X simply adds one to X. The assembler allows you to say INX (for INcrement X) which is a little easier to remember. There is no instruction ADDX. The $400 is the address of the screen (where "$" means a base 16 number) and the $200 is the length of the screen. The lines CMPX and BLO are read: "Compare X with hex 200 and branch if it is lower to LOOP." This means that the STA instruction will be executed $200 times (which is 512 times in base 10). This is once for each character on the screen.

At this point you probably know enough 6809 code to start reading the other example programs given throughout this manual. The best way to learn is to try to make modifications to working programs. For an intensive course in 6809, get a disassembler and try to figure out disassembly listings of programs such as the Basic interpreter ROM.

BASIC ROM ROUTINE ADDRESSES

These are some calls to the Basic ROM in the Color Computer for I/O. Except for Poll Keyboard, these routines expect the Direct Page Register to contain zero. These routines expect certain variables to exist in Basic's variable area from 0000 through 03FF, so beware of where your programs overwrite.

JSR [$A000] Poll keyboard. Returns A=0 if no key has been pressed since last time it was called; otherwise A will contain the ASCII code for the key. Shift-zero is checked for and lowercase is adjusted accordingly. Does not use the Direct Page Register. Condition codes will be zero if A=0, so the call may be followed by a BEQ statement.

JSR [$A002] Output character from A register. Location $006F should contain a unit number (0=screen, etc.) If the unit is -1 or greater than zero, the unit must have previously been opened or a Basic error will occur.

JSR [$A00A] Read joysticks. The four resulting values can be found in $015A through $015D. Each value will be in the range $00 through $3F.

JSR $A171 Input character from unit specified in $006F.

JSR $A176 Same as above, only does not do an ANDA #$7F, so can be used with binary I/O.

JSR $A393 Read line. This gets characters by calling $A171 (so be

sure to set the unit number) and puts them into a buffer at $02DD until a carriage return is read. Backspace, shift-backspace, and clear are checked if the unit is zero (keyboard). If Break is pressed, return will be with carry set.

JSR $A928 Clear text screen, and reset cursor to upper-left corner.

JSR $A951 Generate sound. The B register represents the duration, and the contents of location $008C represents the pitch.

THE "LOAD EFFECTIVE ADDRESS" INSTRUCTIONS

The 6809 instructions LEAX, LEAY, LEAU, and LEAS are some of the most confusing instructions to the beginning 6809er. They are powerful instructions, and yet are also used for mundane operations such as incrementing the X register.

The simplest and commonest use of LEA is to add a value to an index register. For example, LEAX 2,X will add two to the X register, and LEAY -5000,Y will subtract 5000 from the Y register.

The statement LEAS 2,S will pop two bytes from the stack. This could be used to remove an unwanted return address or to clean off some data which was pushed.

The statement LEAY 0,X will add zero to what is in X and put the result in Y; this is the same as TFR X,Y. We could also say LEAY 5,X and set Y equal to five more than X.

The statement LEAX LABEL,PCR is the same as LDX #LABEL except that the LEAX form is the same regardless of where the assembler is told to assemble the program. This is called Position Indepent Code, and enables a program to run even if it is offset-loaded into a different part of memory. See the section on P.I.C. in this manual.

In general, LEA is used with some type of indexed addressing but loads the address instead of what is at the address. For any indexed mode instruction such as LDA 5,S we could instead say:

```
LEAX 5,S     GET ADDRESS OF OPERAND
LDA 0,X      DO THE ACTUAL OPERATION
```

This would be of some benefit if we were to then do many operations on that same byte.

TIMING LOOPS ON THE COLOR COMPUTER

It is often necessary to write programs in which a certain routine must execute at a certain time or after a certain delay. Any program dealing with serial I/O, sound output, interactive graphics, or mechanical delays must involve timing or synchronizing loops.

The processor speed on the Color Computer is 0.895 MHz. Therefore a two cycle instruction such as NOP takes 1.790 microseconds. A one millisecond timing loop is:

```
        LDX #111
        NOP
        NOP
LOOP    LEAX -1,X    (Same as DEX)
        BNE LOOP
```

where the NOPs are there simply to bring the total up to exactly 1000 uSec.

It is possible to synchronize to the 60 Hz vertical refresh rate. The following program allows a section of code to be done sixty times each second:

```
        LDA #$34      Interrupts Off
        STA $FF03     To PIA control register
        LDA $FF02     Clear Flag
LOOP    LDA $FF03     Check control register
        BPL LOOP      Wait til refresh has occurred
        LDA $FF02     Clear Flag
* Do whatever you want in here
        BRA LOOP      Go sync again
```

The following routine synchronizes to the 63 uSec interrupt. Due to the high speed of this signal, the SYNC instruction is used.

```
        ORCC #$50     Inhibit IRQ and FIRQ
        LDA #$35      Interrupts enabled to processor
        STA $FF01     To PIA control register
        LDA $FF00     Clear Flag
LOOP    SYNC          Wait for inhibited IRQ
        LDA $FF00     Clear Flag
* Do whatever you want in here
        BRA LOOP      Go sync again
```

For a very short delay, try the MUL instruction. It takes 11 cycles (about 10 uSec) in only one byte. Just remember that it does affect the D register and the C and Z bits.

INPUT / OUTPUT ON THE COLOR COMPUTER

This section contains several examples of input and output on the serial I/O port, the sound generator, and the joystick buttons. For further information about I/O, refer to Radio Shack's Color Computer Technical Reference Manual.

```
*  READ SERIAL INPUT LINE
       LDA $FF22       PIA 2, B SIDE
       ANDA #1         GET ONLY THE SERIAL INPUT LINE
       BEQ LABEL       BRANCH IF LINE IS ZERO, "SPACE", +12V
```

```
*  WRITE TO SERIAL OUTPUT LINE
        LDA $FF20       OLD OUTPUT VALUE
        ORA #2          SET SERIAL OUTPUT TO 1, "MARK", -12V, IDLE
        STA $FF20       SET SERIAL OUTPUT WITHOUT CHANGING DAC

        LDA $FF20       OLD OUTPUT VALUE
        ANDA #$FF-2     SET SERIAL OUTPUT TO 0, "SPACE", +12V
        STA $FF20       SET SERIAL OUTPUT WITHOUT CHANGING DAC

*  THIS PROGRAM PLAYS A TONE ON THE SPEAKER, BY
*  GENERATING A "SAWTOOTH" WAVE WHICH RUNS THROUGH EACH
*  POSSIBLE VALUE ON THE DAC.
        LDA $FF23       CONTROL REGISTER B
        ORA #8          ENABLE SOUND OUTPUT
        STA $FF23       PUT BACK TO CONTROL REGISTER
        LDA #2          LEAVE SERIAL OUTPUT HIGH SO PRINTER QUIET
LOOP    STA $FF20       OUTPUT TO DAC
        ADDA #4         NEXT VALUE
        TFR X,X         DELAY, TO LOWER THE PITCH
        BRA LOOP        OUTPUT THE NEXT VALUE

*  THIS PROGRAM READS THE JOYSTICK BUTTONS.  IT BRANCHES
*  TO BUTT1 IF THE RIGHT BUTTON IS DOWN, AND TO BUTT2 IF
*  THE LEFT BUTTON IS DOWN.
        LDA #$FF        ALL LINES HIGH
        STA $FF02       TO KEYBOARD OUTPUT; DISABLE KEYS
        LDA $FF00       GET JUST THE JOYSTICK BUTTONS
        BITA #1         RIGHT BUTTON
        BEQ BUTT1       IF RIGHT BUTTON DOWN
        BITA #2         LEFT BUTTON
        BEQ BUTT2       IF LEFT BUTTON DOWN
```

WRITING SOURCE IN BASIC

It is sometimes useful to write a Basic program for generating parts of the source file for an assembly language file. For example, the Basic program below generates a series of FCB instructions which contain a sine table such as might be used in a music program. The file which the program creates can be edited in the normal fashion, and could be appended (using an editor) to a file containing the rest of the assembly source.

```
1000  REM GENERATE SINE TABLE
1010  OPEN"O",#1,"SINETAB/TXT"
1030  FOR I=0 TO 255
1040  X=127*(1+SIN ( I/256 * 2*3.1416 ) )
1050  PRINT #1," FCB ";INT(X)
1060  NEXT I
1070  CLOSE
1080  END
```

EMBEDDING MACHINE LANGUAGE IN BASIC PROGRAMS

It is often useful to create lines in a Basic program which contain machine language. This may be done as long as the machine language contains no zero bytes (and is P.I.C.). To facilitate this type of programming, the assembler has an optional error message which warns whenever a zero byte has been generated. This error message is enabled by placing the psuedo-op "NOZER" in the source program.

As an example, suppose we are writing a Basic program and we want to reverse scroll the screen; that is, to copy everything to the line below. You could just write:

```
9000 FOR I=&H5FF TO &H420 STEP -1: POKE I,PEEK(I-32): NEXT I
```

The only problem is that this takes about five seconds. We'd rather do it in about the same length of time as scrolling up, which is a matter of milliseconds. Therefore, we will write the scrolling program in assembly language and copy it right into a Basic statement.

Step 1: Write the Basic program using this statement:

```
100 DEF USR1=PEEK(47)*256+PEEK(4
8)+34:REM XXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

The exact spacing is important; otherwise you'll have to recalculate that number 34. The number of X's should simply be enough that the program will fit; feel free to add more. Using this method, the size of your program is limited by the input buffer to about 200 bytes.

Step 2: Write the assembly program. The reverse-scroll example is given as sample program EX-SCROLL. Make sure that it's position independent code (see the section on this). Use the ORG 0 statement to make it easier to figure out the offset when you go to load it.

Step 3: Load the Basic program and find the address of the first "X". Do this with a machine language monitor such as DCBUG. An "X" is 58 in ASCII; look for a group of them. Hint: Locations 0019 and 001A point to the start of the Basic program.

Step 4: Load the machine language program into the Basic statement. Use the address you found above as the offset in the LOADM command.
Example: LOADM"EXSCROLL",&H3222
When you next list your Basic program, it will be considerably uglier, but that's OK. Don't try to use an ASCII save (SAVE PRGM,A) and don't try to EDIT the line which contains the machine language (even to remove the excess X's). Feel free to edit, insert, and delete any other lines in the program, and to save or do anything else with it.

Step 5: Use it. We defined the line to be USR1, so a statement such as A=USR1(A) will execute it. The A's are dummies unless the embedded program uses these parameters. If you want to use parameters, see the chapter of Radio Shack's Extended Basic manual called Machine Language Routines.

```
PAGE 0001 THE MICRO WORKS

0001 0E00                       NAM EX-SCROL

                        *  THIS PROGRAM IS DESIGNED TO
                        *  BE EMBEDDED IN A BASIC
                        *  PROGRAM.  IT REVERSE-SCROLLS
                        *  THE TEXT SCREEN.

0002 0E00                          ORG 0            MUST BE P.I.C.

0003 0000                          NOZER            NO ZEROES ALLOWED

0004 0000 8E05E0                   LDX #$600-$20    LAST LINE
0005 0003 EC83          LOOP       LDD ,--X         GET TWO BYTES
0006 0005 ED8820                   STD $20,X        MOVE THEM DOWN
0007 0008 8C0402                   CMPX #$402       START + 2
0008 000B 24F6                     BHS LOOP         LOOP TILL DONE

                        *  JUST FOR FUN LET'S CLEAR THE
                        *  TOP LINE, TOO:

0009 000D CC6020                   LDD #$6020       A=SPACE, B=$20
0010 0010 A780          CLLOOP     STA ,X+          STORE & INCREMENT
0011 0012 5A                       DEC B            COUNT DOWN SPACES
0012 0013 26FB                     BNE CLLOOP       LOOP TILL DONE

                        *  NOW WE LEAVE.  WE DON'T NEED
                        *  TO RESTORE ANY REGISTERS

0013 0015 39                       RTS


NO ERRORS FOUND
```

ASSEMBLING 6800 PROGRAMS

The 6809 processor in the Color Computer is, according to the designers at Motorola, upward compatible on a source code level with its predecessor, the 6800. This means that any 6800 program may be assembled with this assembler and run.

In actuality, the converting of 6800 programs to run on a 6809 is an art. This section will give you enough information to make most programs run, but cannot make the process foolproof.

Many 6800 statements do not need to be converted. The statement CBA (Compare B to A), for example, which generates a single instruction on a 6800, will generate a "PSHS B; CMPA ,S+" in this assembler, which accomplishes the same thing. The only thing to watch out for is a 6800 program which does not keep a stack; if the S register is used as an index register then the converted CBA will alter a byte of data. Such a program, however, is not worth converting.

Some of the 6800 statements are even worth using in new 6809 programs. These are CLC, CLI, CLV, DEX, INX, SEC, SEI, and SEV. They are much clearer than their 6809 counterparts. They are accepted by any 6809 assembler which conforms to Motorola's standards and there is no reason not to use them.

The following problems will creep in when cross-assembling:

The condition codes are not set identically. The CPX instruction on the 6800 (which assembles as a CMPX) does not affect the carry bit. If carry is set as a flag before a CPX the 6809 version will not work.

The TST (test) on the 6809 does not clear carry the way it did on the 6800. This is generally not a problem, but the sequence TST VAR1 / BHI LABEL will not work.

Right shifts on the 6809 do not affect the overflow bit, so the sequence LSR A / BVS LABEL will not work. Such a test would, however, be rare.

Use of the H-flag for other than its intended purpose ("DAA") may not yield identical results. And, of course, any assumption by a 6800 program that E and F bits are always one will not be true.

The stack is the other big problem. Any program which deals with an interrupt stack frame (such as a monitor like ABUG) will not cross assemble since the order and size of the frame is different. Any program that contains an LDS or STS is suspect.

The 6800 instruction TXS was actually a "transfer X to S and decrement"; TSX likewise incremented. Although this could have easily been incorporated in the cross-assembled code, it is more accurate to leave out the decrement/increment. The reason is this: The 6800 stack pointer pointed at the next free byte; the 6809 stack pointer points at the first used byte. The instruction TSX will point the X register at the last byte pushed if it increments on the 6800

but does not increment when cross-assembled to the 6809.

The consequence of the considerations in the paragraph above is simply this: A 6800 program which contains only TSX and TXS instructions is safe to cross assemble. Any LDS or STS instruction may need work. An LDS #$xxxx (such as is commonly found) may have its value increased by one, such as from $3FFF to $4000. An STS SAVSTK followed later by LDS SAVSTK is all right, but if LDX SAVSTK appears it is guaranteed not to work.

The last problem is in software timing loops. These have to be looked at anyway since they depend on processor speed (0.895 MHz on the Color Computer). Many 6809 instructions take more cycles than the 6800 equivalents. Load A extended takes five instead of four cycles. Compare B to A (which crosses to the PSH/CMP shown above) will take twelve instead of two cycles. Conditional branches, however, only take three cycles, so the loop

```
        LDX #COUNT
LOOP    DEX
        BNE LOOP
```

will still take eight cycles as it did on the 6800, even though both of its instructions take a different number of cycles each. Most other timing loops will require careful counting.

In summary, to cross assemble a 6800 program, look for these items:

1. CPX - is carry tested afterwards?
2. TST - is carry tested afterwards?
3. right shifts - is overflow tested?
4. any unusual operations on the condition codes
5. all LDS and STS instructions
6. any software timing loops

If all of these points are checked, most 6800 programs should give little trouble in cross assembly.

6800 CROSS MNEMONICS

The following mnemonics are supported by the assembler so that code written for the 6800 processor can be assembled.
Some of these instructions (such as INX or CLI) are clearer and easier to use than their 6809 equivalents and may be used instead.

6800 inst	Generated code equivalent
ABA	PSHS B; ADDA ,S+
CBA	PSHS B; CMPA ,S+
CLC	ANDCC #$FE
CLI	ANDCC #$EF
CLV	ANDCC #$FD
DES	LEAS -1,S
DEX	LEAX -1,X

```
INS         LEAS 1,S
INX         LEAX 1,X
SBA         PSHS B; SUBA ,S+
SEC         ORCC #$01
SEI         ORCC #$10
SEV         ORCC #$02
TAB         TFR A,B; TST A
TAP         TFR A,CC
TBA         TFR B,A; TST B
TPA         TFR CC,A
TSX         TFR S,X
TXS         TFR X,S
WAI         CWAI #$FF
```

Also included are some 6800-like mnemonics which relate to Fast Interrupts, which the 6800 does not have. They may be clearer to use than their 6809 equivalents and are included for this reason.

```
CLF     ANDCC #$DF      Clear Fast Interrupt inhibit bit
CLIF    ANDCC #$AF      Clear both interrupt inhibit bits
SEF     ORCC #$40       Set Fast Interrupt inhibit bit
SEIF    ORCC #$50       Set both interrupt inhibit bits
```

POSITION INDEPENDENT CODE

The 6809 programmer should understand position independent code, as it is a powerful tool which is readily available on this processor. Although it is possible to write P.I.C. on other CPU's such as the 6800, it is seldom easy. On the 6809, however, there is almost no excuse not to.

Position independent code (or P.I.C.; relocatable code; or run-anywhere code) is code which can be moved to anywhere in the memory space AFTER ASSEMBLY and correctly run. For example, a P.I.C program which runs at $1000 could be block moved up to $2000 and it would run just as well. If it contained any statement such as JMP LABEL (where LABEL is within the program) then it would not be P.I.C., since it would jump to the wrong address if the code were moved. Writing P.I.C., then, is basically avoiding certain statements and addressing modes which do not work when the object code is moved.

The most obvious rule is this: Replace JMP with LBRA (long branch), and replace JSR with LBSR (long branch to subroutine). In itself, however, this is not enough; a program which is only partially P.I.C. is not P.I.C. at all and is merely slower than the version which contains JMPs. Therefore, the discussion below is included in order to allow you to write code which is entirely P.I.C.

All symbols should be classified as to whether they are in the program or absolute; that is, whether or not they will move when the program does. A label is in the program; so is a variable which is declared as part of the program (which makes it non-ROMable, but that's another story). A variable in page zero is absolute. A routine in the BASIC ROM is absolute. A symbol defined by an equate statement is

absolute, unless there is a program label on the right side of the equate. A variable in a stack frame is absolute (which is yet another story).

Any reference to a symbol in the program must be relative; any reference to an absolute symbol must be absolute. A jump to a label in the program must use BRA or LBRA, while a jump to a routine in ROM cannot use LBRA and must use JMP.

A reference to a variable or constant which is part of the program must use the PCR addressing mode. LDA XYZ,PCR has the same effect as LDA XYZ except that relative addressing is used. On the other hand, the PCR mode may not be used when referencing variables which are defined by absolute symbols. Remember the adage, "Everything not mandatory is prohibited."

The statement "LDX #TABLE" assembles to an absolute address which must not be used if TABLE is within the program. The statement to be used instead is "LEAX TABLE,PCR". A thorough understanding of the LEA statement in its many forms is necessary for a good 6809 programmer.

The statement "CMPX #TABLE" is harder to replace. It is often used to check to see if we have finished stepping through a table. It is best avoided by various means which depend on the program in question, but if a quick fix is needed, this sort of thing will work:

```
        LEAX ENDTAB,PCR
        PSHS X
        LEAX TABLE,PCR
LOOP    ADDA ,X+    (or whatever)
        CMPX 0,S
        BLO LOOP
        PULS X      (clean up stack)
        ...
TABLE   FCB 1,2,3,4,5
ENDTAB  EQU *
```

In the example above, by the way, the PULS X could have been replaced by a LEAS 2,S. However, it is clearer and less conducive to errors to match a pull with a push.

One source of problems is the FDB statement. Any time a label in the program appears in an FDB statement, an absolute constant may be generated. A typical example is the jump table, wherein this sort of thing goes on:

```
        LDX #TABLE
        ASL B
        LDX B,X
        JMP 0,X
TABLE   FDB PLACE1,PLACE2,PLACE3,...
```

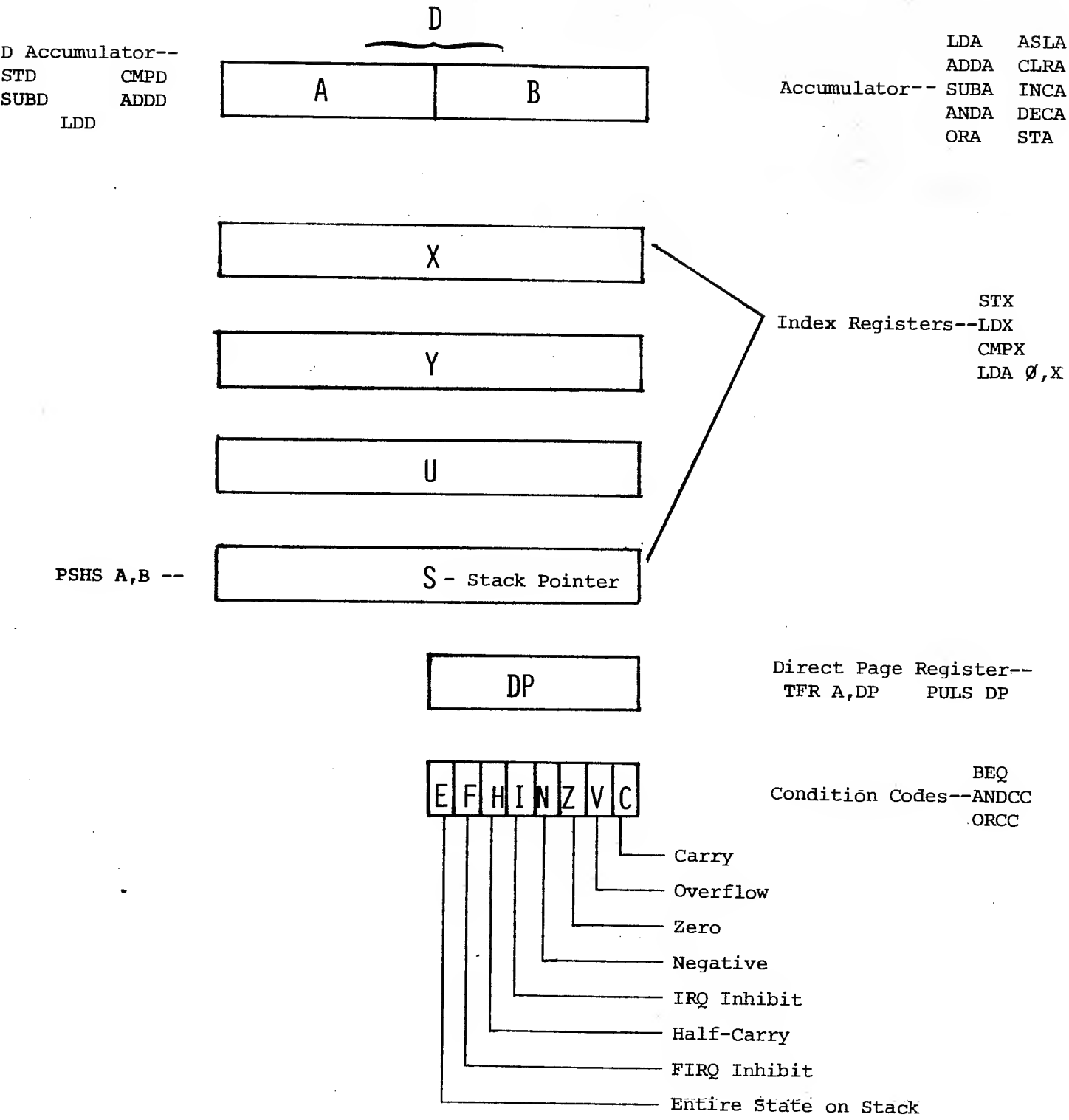
The LDX becomes an LEAX, of course, but what about the FDB? One solution is to add onto X (just before the jump) some relocation

constant which can be obtained by the strange-looking statement LEAX 0,PCR. Another solution is this:

```
            LEAX TABLE,PCR
            ASL B
            ABX
            LDD 0,X
            JMP D,X
    TABLE   FDB PLACE1-*,PLACE2-*,PLACE3-*,...
```

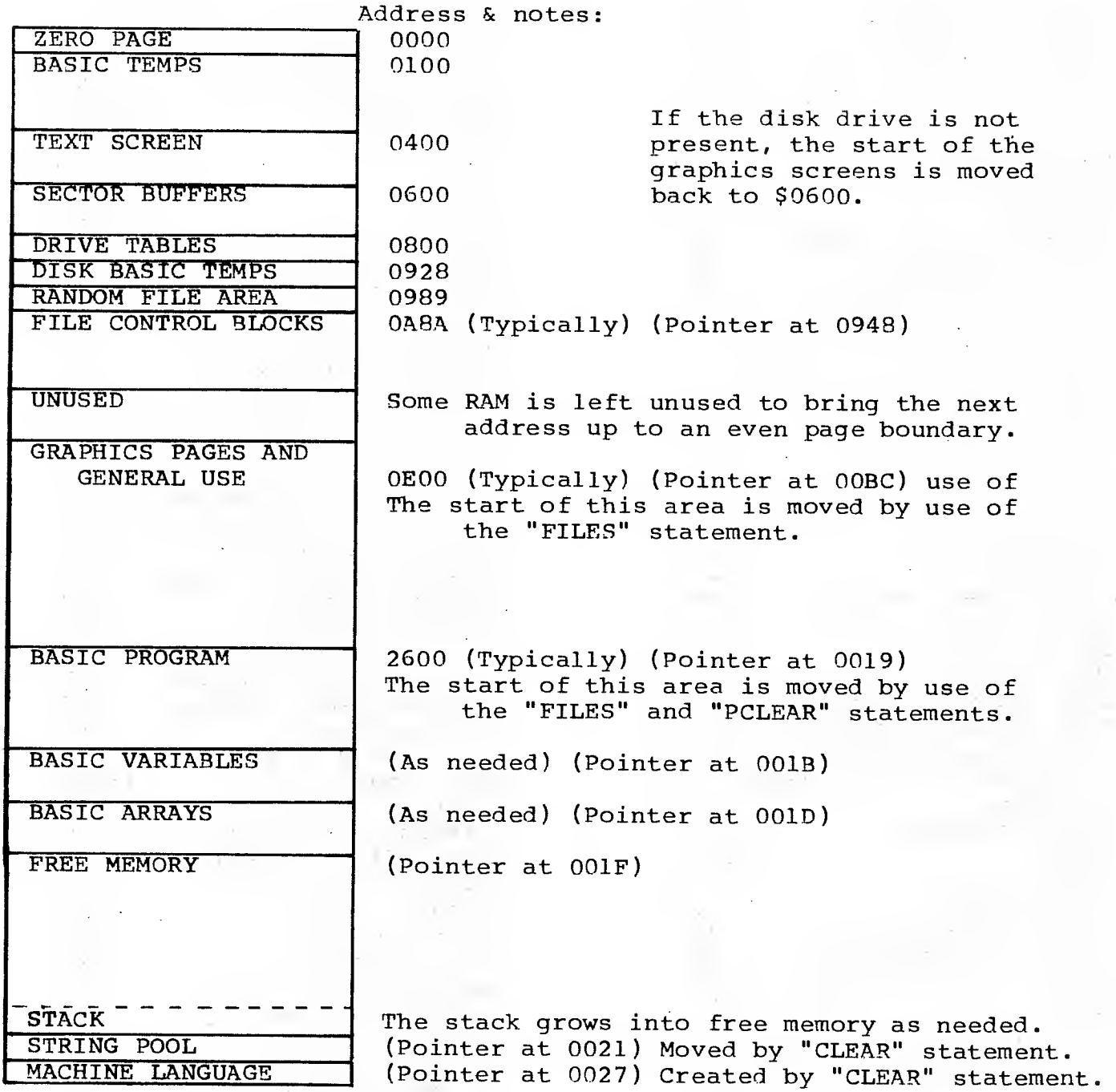
Who says you can't have a "JMP" in P.I.C.?

PROGRAMMER'S MODEL OF THE 6809



Above are examples of typical instructions which access each register. Not shown is the Program Counter, the use of which is inherent in programming.

MEMORY MAP OF THE COLOR COMPUTER DISK SYSTEM



LAST RAM ADDRESS = 3FFF IN A 16K SYSTEM,
7FFF IN A 32K SYSTEM.

EXTENDED BASIC:	8000-9FFF
BASIC:	A000-BFFF
DISK BASIC:	C000-D7FF
I/O:	FF00-FF5F
SYSTEM CONTROL:	FFC0-FFDF

I N D E X